(1)

Technical Report 1053

Dependency-Directed Localization of Software Bugs

Ron I. Kuper

MIT Artificial Intelligence Laboratory

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR 1053	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Dependency-Directed Localization of Software Bugs		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ron I. Kuper		8. CONTRACT OR GRANT NUMBER(s) N00014-88-K-0487
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE May 1989
		13. NUMBER OF PAGES 74
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

debugging
programmer's apprentice

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A software bug manifests itself as a violated specification. Debugging is the process that culminates in repairing a program so that it satisfies its specification. An important part of debugging is *localization*, whereby the smallest region of the program that manifests the bug is found. The Debugging Assistant (DEBUSSI) is a system that localizes bugs by reasoning about logical dependencies. Via queries to the user and automated deduction, DEBUSSI manipulates the assumptions that underlie a

Block 20 cont.

bug manifestation, eventually localizing the bug to one particular assumption. At the same time DEBUSSI incrementally acquires specification information, thereby extending its understanding of the buggy program. As part of the Programmer's Apprentice, DEBUSSI will be useful for validating partial designs, as well as for testing fully implemented code.

Dependency-Directed Localization of Software Bugs

by

Ron I. Kuper

May 10, 1989

Abstract

A software bug manifests itself as a violated specification. Debugging is the process that culminates in repairing a program so that it satisfies its specification. An important part of debugging is *localization*, whereby the smallest region of the program that manifests the bug is found. The Debugging Assistant (DEBUSSI) is a system that localizes bugs by reasoning about logical dependencies. Via queries to the user and automated deduction, DEBUSSI manipulates the assumptions that underlie a bug manifestation, eventually localizing the bug to one particular assumption. At the same time DEBUSSI incrementally acquires specification information, thereby extending its understanding of the buggy program. As part of the Programmer's Apprentice, DEBUSSI will be useful for validating partial designs, as well as for testing fully implemented code.

This publication is a revised version of *Automated Techniques for the Localization of Software Bugs*, a report submitted to the Department of Electrical Engineering and Computer Science on May 12, 1988 in partial fulfillment of the requirements for the degree of Master of Science.

The research described here was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the following organizations: National Science Foundation under grant IRI-8616644, Advanced Research Projects Agency of the Department of Defense under Naval Research contract N00014-88-K-0487, IBM Corporation, NYNEX Corporation, and Siemens Corporation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the policies, expressed or implied, of these organizations.

Contents

Chapter 1

Introduction

One of the most dreaded tasks in programming is debugging unfamiliar code. A methodical approach is the only way to manage the complexity of this task. First, determine which subroutines actually contribute to the program's incorrect output. Then choose a subroutine and consult its source code and documentation to determine what it is supposed to do. Finally, compare its expected behavior with its observed behavior; where they disagree is where the bug can be found.

In this approach to bug localization, the focus is on reasoning about a program's structure and behavior. The control and data flow structure of a program gives rise to a network of dependencies between its parts. These dependencies determine how a bug ultimately manifests itself, so they provide leverage to the task of bug localization. Furthermore, approaching bug localization at the level of dependencies allows idiosyncratic details of a program's implementation to be ignored, therefore allowing debugging techniques to be applied to partially implemented programs.

The Debugging Assistant (DEBUSSI[1]) is an experimental system that helps a programmer localize bugs. DEBUSSI finds an initial set of suspects by reasoning about the dependencies that arise from data flow and control flow. With well-chosen queries to the user, it incrementally acquires missing specification information, until it has enough information to rule out all but one suspect. If this one remaining suspect has internal structure, localization continues recursively on its parts.

DEBUSSI is a part of the Programmer's Apprentice project [19], both in terms of its philosophy and and its technology. DEBUSSI is intended to work side-by-side with the programmer as a junior partner. It is implemented using Cake [15], a powerful reasoning system developed in the project.

Studying debugging contributes to an understanding of general problem solving principles. We believe that the techniques that people use in debugging programs are similar to those in other problem solving tasks, such as hardware troubleshooting and medical diagnosis. Understanding how to debug unfamiliar code is especially

[1]Apologies to the late composer.

revealing of general problem solving abilities, because it requires reasoning in the face of limited experience. Debugging unfamiliar code provides insight into "first-principles" reasoning, as opposed to experience-based reasoning.

1.1 The Approach Taken by DEBUSSI

In any discussion of automated program debugging, several major issues must be addressed. This section will discuss these debugging issues, and will characterize DEBUSSI's approach in terms of these issues.

A first issue is how the debugger represents programs and specifications. Predicate logic, parse trees and graph formalisms are examples of common program representations. A good representation should be programming language independent and should easily scale to large programs.

DEBUSSI represents programs using a hybrid of graphical and logical formalisms known as the Plan Calculus [2, 13, 16]. A *plan* is a graph whose nodes represent operations in a program, and whose arcs represent data flow and control flow. Plans also have logical annotations to represent preconditions, postconditions and data invariants.

Plans are programming-language independent. They abstract away from such language-specific details as variable binding constructs and unnecessary sequentialization of control flow. Plans are also hierarchical. They allow computations to be viewed as "black boxes" whose internal structure may or may not be important.

A second issue in debugging is what part of the task is actually performed by the debugger. Debugging can be viewed as being composed of the following sequence of steps: test generation, bug detection, bug localization, bug understanding and bug repair. Various systems focus on one or more of these steps.

DEBUSSI performs *bug localization*, isolating an error to a particular function call or section of code. Bug localization becomes critically important in programming-in-the-large, where programs can be millions of lines long. In such programs, the sheer quantity of code makes bug localization a very labor intensive task.

A final debugging issue is the type of knowledge and deductive capabilities that are brought to bear. Debugging systems may use several kinds of programming knowledge. For example, a system may have a library of "bug patterns," which are matched against a buggy program to find errors. A system may also have some knowledge about program design. A debugger empowered with this kind of knowledge can compare its design decisions with the programmer's, with the expectation that the bug will be found in some bad decision.

Debugging systems also employ a variety of deductive methods. One commonly used method is to rewrite programs via transformation rules, eventually making the program syntactically equivalent to its buggy counterpart. Another method is to represent the program via logical formulae, and use automated theorem proving to demonstrate its correctness or incorrectness.

DEBUSSI utilizes knowledge of program's specifications, control flow and data flow. Control flow and data flow information is obtained from the program's source code. Specification knowledge may be provided *a priori*, or may be obtained from the user.

DEBUSSI's programming knowledge is represented as logical predicates. Deduction on these predicates is done by Cake [15], a layered reasoning system developed for the Programmer's Apprentice. An important aspect of the deductive capability of Cake is it's use of truth maintenance techniques to record logical dependencies.

1.2 Overview of DEBUSSI's Operation

Figure 1.1 depicts DEBUSSI's overall architecture. There are two major components to the system: the reasoning system, Cake; and the debugging assistant, DEBUSSI. Cake is an existing system in the Programmer's Apprentice project. DEBUSSI is the contribution of this research.

DEBUSSI accepts input in the form of source code. (In the current implementation of DEBUSSI, all programs must be side-effect free.) As shown in the lower left corner of the figure, a program analyzer translates the source code into plans. (In the current implementation of DEBUSSI, source-code analysis is done by hand, but technology exists in the Programmer's Apprentice to automate this process [26].) Recall that the plan is a hybrid representation for the program, combining graphical and logical formalisms. The program analyzer encodes the graphical aspect of the plan as logical formulae, and through these formulae Cake reasons about the run-time behavior of the program.

In addition to program plans, the reasoning system may be given some initial partial specification information. For example, the user (or an automated assistant [3]) can supply an explicit test case. Alternatively, specifications may already exist in the design environment.

If the program is buggy, then its specifications will be in conflict with its behavior. This conflict will manifest itself in the reasoning system as a contradiction. DEBUSSI commences bug localization when a contradiction arises.

Given a contradiction (bug manifestation) DEBUSSI first forms an initial set of suspects via *dependency analysis*. By exploring the dependency structure that leads to the contradiction, DEBUSSI determines the premises that underlie the bug manifestation. The initial set of suspects is determined from this set of premises.

Because DEBUSSI bases its dependency analysis on violated specifications, a bug can be localized only if it can be made to manifest itself. In other words, if some bug is lying dormant in a program, and the user has no test case (or specification) that reveals that bug's presence, DEBUSSI will be unable to localize it. Some of the suspects will be conditional statements, or "splits" in control flow. DEBUSSI

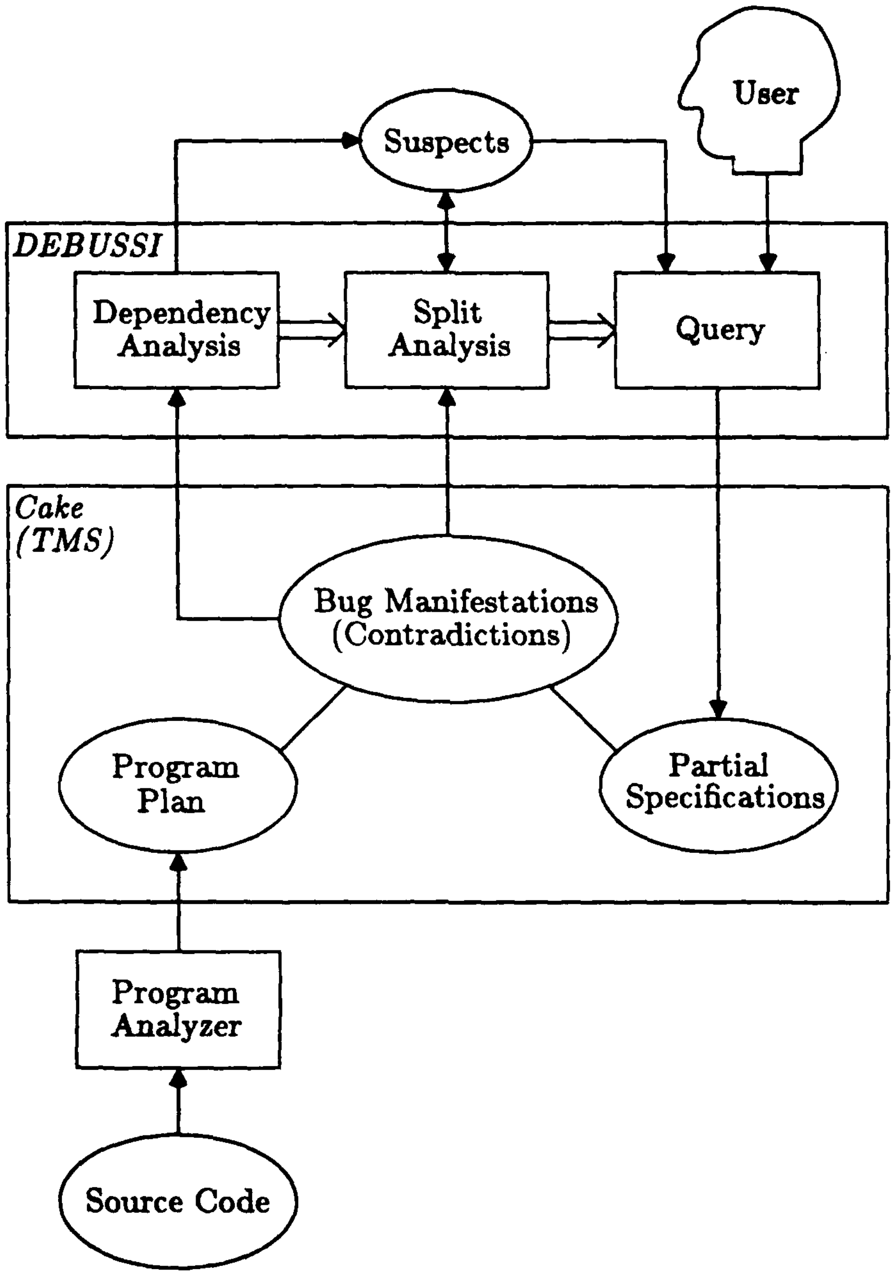


Figure 1.1: Architecture of DEBUSSI

attempts to exonerate splits by performing *split analysis* on each one. In split analysis. DEBUSSI observes how the program's output changes when a split "goes the other way." If changing a split's outcome does not alleviate the bug manifestation, then the split can be exonerated. (See Chapter 3 for more details).

Dependency analysis and split analysis will usually exonerate some, but not all, of the suspects. At this point DEBUSSI requires more specification information, so it queries the user about one of the remaining suspects. In choosing which suspect to ask about, DEBUSSI balances the need to obtain as much new information as possible with the need to avoid overtaxing the user's mental faculties. Finding the middle ground between these two opposing constraints is done with heuristics.

The new information obtained in the query will result in a change in the logical dependency structure within Cake. The changed dependency structure will often reflect the exoneration of one or more suspects. To determine which suspects may be exonerated as a result of the query, DEBUSSI once again performs dependency analysis on the initial bug manifestation.

DEBUSSI continues iterating through the steps of dependency analysis, split analysis and querying until it finds itself with one suspect or no suspects. If one suspect remains, and it represents a call to a system primitive (such as `Car` in Lisp), then DEBUSSI concludes that the bug is due to an incorrect use of that primitive.

If one suspect remains, and it represents a call to a user-defined function, then DEBUSSI will attempt to further localize the bug to some point within that function call. In this case, DEBUSSI "zooms in" on the function call by expanding its definition inline, and continues by recursively debugging the expanded definition.

If DEBUSSI is unable to localize the bug after zooming in, it ends having no suspects. In this case, DEBUSSI concludes that the bug is at some undetermined place within the enclosing function call. This situation typically arises when the bug is due to faulty data flow, such as incorrect argument order for a function call. (Another way for DEBUSSI to come up with zero suspects is when the program has more than one bug. Chapter 3 addresses this issue in detail.)

1.3 DEBUSSI As Part of the Programmer's Apprentice

DEBUSSI shares three key philosophical features with the Programmer's Apprentice. First, it is an active tool, cooperating with the programmer as an assistant. This is demonstrated by DEBUSSI's query mechanism, which does some things by itself, and gets assistance from the user for others.

Second, DEBUSSI adds to, rather than replaces, existing programming environments. For example, we expect other systems [3] to provide DEBUSSI with test cases that will allow the automatic testing of potentially buggy code.

Third, DEBUSSI follows the approach of incremental automation, meaning that its power will advance with improvements in the underlying technology. One such technological advance would be the inclusion of a program's design history into the debugging environment. The design history could supply knowledge of which functions have been tested the least or edited most recently, knowledge that could assist the task of bug localization.

DEBUSSI uses Cake [15], a technological resource of the Programmer's Apprentice project. Cake is a layered reasoning system, supporting reasoning about functions, sets, frames and plans. It includes a truth maintenance system, which makes it easy to deal with the changing and possibly contradictory information that is inherent in the task of debugging.

1.4 Organization

Chapter 2 presents several transcripts illustrating the key ideas underlying DEBUSSI's localization techniques. These transcripts are actually runnable in the current implementation of DEBUSSI. Chapter 3 describes the localization algorithm in detail and discusses the heuristics employed in query selection. Chapter 4 discusses related work, including a comparison of the domains of software bug localization and hardware troubleshooting. Chapter 5 addresses limitations of the localization algorithm and proposes future work.

Chapter 2

Localization Scenarios

This chapter contains three scenarios illustrating DEBUSSI's bug localization techniques. The first scenario demonstrates how DEBUSSI localizes a non-fatal bug. The second scenario demonstrates the localization of a fatal bug, i.e., a bug that manifests itself by "crashing" the program. The final scenario demonstrates that DEBUSSI's localization techniques are general enough to be applied to partially implemented programs.

There are several important recurring themes in this chapter. One theme is the range of DEBUSSI's applicability. DEBUSSI can localize bugs in fully implemented programs by running them on concrete data and comparing observed and expected results. Alternatively, DEBUSSI can debug partially implemented programs by attempting to prove that an implementation satisfies a set of partial or complete specifications.

A second important theme is DEBUSSI's use of heuristics, both in choosing queries and exonerating suspects. Heuristics are useful in choosing queries because the structure of a program may not provide enough insight into which parts of the program are most likely to be buggy. Heuristics are useful for exonerating suspects because they provide shortcuts that can reduce the workload of the automated reasoning system.

A third theme is the way DEBUSSI uses queries to incrementally acquire partial specification information. The underlying principle is that if a programmer recognizes that some part of a program is buggy, he can often provide a violated specification that characterizes the bug. For example, if a programmer indicates that some function argument has the wrong value, he is asked to provide a description of the correct value.

2.1 Terminology

A *bug* is an error in the structure of a program. One common type of bug is the incorrect use of a programming language primitive, for example, using **Cons** to concatenate two lists instead of Append. Another common type of bug is incorrect dataflow within a function, for example, calling `(Member The-List 'X)` instead of `(Member 'X The-List)`.

Programmers usually find bugs by noticing violated specifications, or *bug manifestations.* A common type of bug manifestation is the failure of a simple test case, where a program produces an incorrect output for a particular input. Another common bug manifestation is abnormal termination, or "crashing." A third type of bug manifestation is nontermination, or "infinite looping."

There is an important distinction between bugs and bug manifestations. Bugs are identified by noticing incongruities in a program's source code. Bug manifestations are identified by noticing inconsistencies in a program's run-time behavior. In other words, if a program is viewed as a black box with input and output ports, then a bug is a structural error found inside the box. A bug manifestation is an error observed on the box's ports.

DEBUSSI performs *bug localization* by finding the innermost dynamic function call that manifests a bug. To better understand the notion of "innermost," suppose that the function Main manifests a bug, and Main calls Func (among others). Initially, DEBUSSI localizes the bug to Main, but if it can show that Func also manifests a bug, then the bug is further localized to Func.

A crucial characteristic of DEBUSSI's localization technique is that it localizes bugs to function *calls*, not function *definitions.* In the previous example, if Main had called Func twice, then DEBUSSI would consider each call to Func as an independent suspect. After the bug has been localized to a particular call, the user can choose between replacing the call with a different function, or redefining Func.

In the process of localizing a bug, DEBUSSI maintains a set of *suspects*, which is the set of potentially buggy function calls. This set is narrowed down by *exonerating* one or more suspects, i.e., removing them from the current set of suspets. Repeated exoneration will either narrow the suspect set down to a single *culprit*, or will rule out all of the suspects (see Chapter 3).

2.2 The User Interface

The scenarios below present an idealized version of DEBUSSI's user interface. The differences between the ideal and actual interfaces to the running prototype are mostly aesthetic and require no changes to DEBUSSI's automated techniques. This section provides an overview of the interface and explains the presentational conventions.

Sample Frame

```
Function call: (ReEnqueue 'Job15 '(Job15 Job18))
Returned (incorrect) value: (Job15)
```

```
(Defun ReEnqueue (Item Queue)
  (If (Queue-Empty? Queue)
      (Make-Queue Item)
    (Enqueue Item (Queue-Delete Item Queue))))
                                                        CODE WINDOW
```

```
Suspects due to dependency analysis:
  Queue-Empty Queue-Delete Enqueue
Split analysis exonerates:
  Queue-Empty?

Function call: (Queue-Delete 'Job15 '(Job15 Job18))
Returned value: Nil

Was Queue-Delete called correctly? Yes
Did Queue-Delete return the right result? No
                                                   INTERACTION WINDOW
```

The figure above is a sample frame illustrating DEBUSSI's user interface. The user and DEBUSSI are tracking down a bug within ReEnqueue, a procedure that takes an item and a queue and returns a new queue with the item at its head.

At the top of the figure is the code window. It contains a source-code listing of the procedure that is the current focus of attention. The calls to Enqueue and Queue-Delete are displayed in a bold typeface, indicating that they are suspects. The call to Queue-Empty is underlined, indicating that it has just been exonerated. Queue-Delete has a box around it to indicate that it is the suspect currently being queried about.

Below the code window is the interaction window. It is the locus of communication between DEBUSSI and the user. DEBUSSI keeps the user abreast of its actions via messages displayed in this window.

The last few lines in the interaction window are a simple query about the boxed call of Queue-Delete. The user's response to the query is displayed in bold.

The only difference between DEBUSSI's actual interface and this hypothetical interface is the presence of the code window. Since the code window is merely a presentation of the program's source code, its addition does not add any power to DEBUSSI's automated techniques, but does increase its usability.

A practical interface to DEBUSSI would be considerably less verbose than the one presented in this chapter. In particular, any actions taken by DEBUSSI that do not explicitly require user interaction can be done silently, and in this chapter, frames which contain such actions will be indicated as "Not Presented to the User."

2.3 A Program Without Specifications

The first two scenarios demonstrate how DEBUSSI localizes bugs in a program having no specifications. All specification information that DEBUSSI obtains will come from the user. The only initial information available is source code, from which DEBUSSI can determine control and data flow.

The example program is a Common Lisp [23] program that performs unification [27]. It's inputs are two patterns and an environment (an initial list of bindings). Patterns are represented as lists. Variables are represented as lists whose **Car** is a question-mark. Bindings are represented as cons-cells whose **Car** is a variable.

```
> (Unify '(F (? X) (? Y)) '(F 3 4) Nil)
(((? X) . 3) ((? Y) . 4))
> (Unify '(F (? X)) '(G 3) Nil)
NoMatch
> (Unify '(F 1) '(F 1) Nil)
Nil
```

Three examples of calls to Unify are shown above. In the second example, the two patterns given to Unify are incompatible, and Unify returns the special atom NoMatch. In the third example, the two patterns match trivially (i.e., with no bindings), and Unify returns Nil.

The top-level code for the unifier is shown in Figure 2.1. **Unify** recursively compares the two patterns, updating the environment when a variable is encountered in either pattern. **Var-Var-Match** compares two variables and, if necessary, updates the environment after binding one to the other. **Var-Pat-Match** matches a variable with a pattern, making sure that the pattern is compatible with the current environment.

The remaining code in the unifier implementation is shown in Figure 2.2. **Freeof?** verifies that a pattern does not contain any instance or binding of a particular variable. **Lookup** returns the binding of a given variable within an environment, and **Extend** adds a new binding to an environment.

Constant? returns a non-**Nil** value when its argument is an atom, i.e., not a list or variable. **Same-Constant?** compares two constants using **Eq**. **Var?** returns non-**Nil** when its argument is a variable, i.e., a Cons-cell whose **Car** is a question-mark. **New-Binding**, **Binding-Var** and **Binding-Val** are the constructor and accessors for variable bindings.

```
(Defun Unify (P1 P2 Env)
  (Cond ((Eq Env 'NoMatch) 'NoMatch)
        ((Var? P1)
         (If (Var? P2)
             (Var-Var-Match P1 P2 Env)
           (Var-Pat-Match P1 P2 Env)))
        ((Var? P2) (Var-Pat-Match P2 P1 Env))
        ((Constant? P1)
         (If (Constant? P2)
             (If (Same-Constant? P1 P2)
                 Env
               'NoMatch)
           'NoMatch))
        ((Constant? P2) 'NoMatch)
        (T (Unify (Cdr P1)
                  (Cdr P2)
                  (Unify (Car P1) (Car P2) Env)))))

(Defun Var-Var-Match (V1 V2 Env)
  (If (Eq V1 V2)
      Env
    (Let ((B1 (Lookup V1 Env))
          (B2 (Lookup V2 Env)))
      (If (Null B1)
          (If (Null B2)
              (Extend V1 V2 Env)
            (Unify V1 (Binding-Val B2) Env))
        (Unify (Binding-Val B1)
               (If (Null B2)
                   V2
                 (Binding-Val B2))
               Env)))))

(Defun Var-Pat-Match (Var Val Env)
   (Let ((Value-Cell (Lookup Var Env)))
     (If (Null Value-Cell)
         (If (Freeof? Var Val Env)
             (Extend Var Val Env)
           'NoMatch)
       (Unify (Binding-Val Value-Cell) Val Env))))
```

Figure 2.1: Implementation of Unify, Var-Var-Match and Var-Pat-Match.

```
(Defun Freeof? (Var E Env)
  (Cond ((Constant? E) T)
        ((Var? E)
         (If (Equal Var E)
             Nil
           (Let ((B (Lookup E Env)))
             (If (Null B)
                 T
               (Freeof? Var (Binding-Val B) Env)))))
        ((Freeof? Var (Car E) Env)
         (Freeof? Var (Cdr E) Env))))

(Defun Lookup (Var Env)
  (Cond ((Null Env) Nil)
        ((Equal Var (Binding-Var (Car Env))) (Car Var))
        (T (Lookup Var (Cdr Env)))))

(Defun Extend (Var Val Env)
  (Cons (New-Binding Var Val) Env))

(Defun Constant? (X) (Atom X))

(Defun Same-Constant? (X Y) (Eq X Y))

(Defun Var? (X)
  (And (ConsP X) (Eq (Car X) '?)))

(Defun New-Binding (Var Value) (Cons Var Value))

(Defun Binding-Var (Cell) (Car Cell))

(Defun Binding-Val (Cell) (Cdr Cell))
```

Figure 2.2: Miscellaneous functions used by `Unify`.

2.4 Localizing a Non-Fatal Bug

There are two bugs in this implementation of `Unify`. This first scenario will illustrate how DEBUSSI localizes one of the two bugs based on a failed test case.

Frame 1

```
(Defun Unify (P1 P2 Env)
   (Cond ((Eq Env 'NoMatch) 'Nomatch)
         ((Var? P1)
          (If (Var? P2)
              (Var-Var-Match P1 P2 Env)
            (Var-Pat-Match P1 P2 Env)))
         ((Var? P2) (Var-Pat-Match P2 P1 Env))
         ((Constant? P1)
          (If (Constant? P2)
              (If (Same-Constant? P1 P2)
                  Env
                'NoMatch)
            'NoMatch))
      ((Constant? P2) 'NoMatch)
      (T (Unify (Cdr P1)
                (Cdr P2)
                (Unify (Car P1) (Car P2) Env)))))
```

CODE WINDOW

```
> (Unify '((? X)) '((? X)) Nil)
((? X) . (? X))
> (Correct-Output-Is Nil)
```

INTERACTION WINDOW

As the scenario begins, the user has completed writing the code for `Unify` and its related functions and tests the program on a simple test case. (The test case could have alternatively been supplied by an automated system.) The result of the test case is not what the user expected. Unifying `(? X)` with `(? X)` should have returned `Nil`, indicating that the two patterns matched with no substitutions. At this point, the user invokes DEBUSSI by typing `(Correct-Output-Is Nil)`. This command instructs DEBUSSI to begin localizing a bug from a failed test case.

Frame 1.1 (Not Shown to the User)

```
Function call: (Unify '((? X)) '((? X)) Nil)
Returned (incorrect) value: ((? X) . (? X))
```

```
(Defun Unify (P1 P2 Env)
  (Cond ((Eq Env 'NoMatch) 'NoMatch)
        ((Var? P1)
         (If (Var? P2)
             (Var-Var-Match P1 P2 Env)
           (Var-Pat-Match P1 P2 Env)))
        ((Var? P2) (Var-Pat-Match P2 P1 Env))
        ((Constant? P1)
         (If (Constant? P2)
             (If (Same-Constant? P1 P2)
                 Env
               'NoMatch)
           'NoMatch))
        ((Constant? P2) 'NoMatch)
        (T (Unify (Cdr P1)
                  (Cdr P2)
                  (Unify (Car P1) (Car P2) Env)))))
                                                        CODE WINDOW
```

```
Suspects due to dependency analysis:
  Eq Var? Constant? Car Unify Cdr
Split analysis exonerates:
  Eq Var? Constant?
                                                 INTERACTION WINDOW
```

Frame 1.1 shows DEBUSSI forming an initial set of suspects by tracing dependencies from the bug manifestation at the output of Unify. Two factors contribute to this output: the data flow that is used to compute the value, and the conditional control flow that causes the value to be computed.

The output of Unify is computed in the final clause of the Cond statement. This final clause consumes data supplied by the inputs to Unify, and produces its result via calls to Car, Cdr and Unify. Since the output depends on their results, the indicated calls to Car, Cdr and Unify are suspected, because of their contribution to the pattern of data flow.

The final clause of the Cond was executed because all previous tests within the Cond failed. If any of them had succeeded, the output to Unify might have been computed differently. Thus the calls to Eq, Var? and Constant? are suspected because of their contribution to the overall pattern of control flow.

Note that in the interaction window, multiple calls to a function are summarized as a single reference to that function. For example, two calls on Var? are suspected after dependency analysis. Rather than trying to distinguish between these calls in the interaction window, DEBUSSI simply refers to `Var?`, and uses the code window to differentiate between the offending calls.

DEBUSSI reasons about conditional control flow by performing *split analysis*, i.e., exploring what happens when splits "go the other way." (Conditional statements will often be referred to as "control flow splits," or just "splits.") For example, consider the first call to `Eq` in `Unify`. `Eq` returns `Nil`, causing its consequent in the `Cond` statement to be passed over. If `Eq` had been buggy, then replacing it with another predicate that returned `T` would cause its consequent in the `Cond` statement to be executed. If a bug manifestation still arises even after making this quasi-repair, then `Eq` must not have been buggy in the first place. As shown in the frame via the underlined calls, split analysis also exonerates calls to `Var?` and `Constant?`.

Split analysis is essentially the same technique as *constraint suspension* [4] in hardware troubleshooting, since their common purpose is coaxing a reasoning system to create additional dependencies. The underlying reasoning technique in split analysis is "proof by cases," where the cases to consider are the two outcomes of the control flow split.

Frame 2

```
Function call: (Unify '((? X)) '((? X)) Nil)
Returned (incorrect) value: ((? X) . (? X))
```

```
(Defun Unify (P1 P2 Env)
  (Cond ((Eq Env 'NoMatch) 'NoMatch)
        ((Var? P1)
         (If (Var? P2)
             (Var-Var-Match P1 P2 Env)
           (Var-Pat-Match P1 P2 Env)))
        ((Var? P2) (Var-Pat-Match P2 P1 Env))
        ((Constant? P1)
         (If (Constant? P2)
             (If (Same-Constant? P1 P2)
                 Env
               'NoMatch)
           'NoMatch))
        ((Constant? P2) 'NoMatch)
        (T (Unify (Cdr P1)
                  (Cdr P2)
                  (Unify (Car P1) (Car P2) Env)))))
                                                        CODE WINDOW
```

```
Function call: (Unify Nil Nil '(((? X) . (? X))))
Returned (incorrect) value: (((? X) . (? X)))

Was Unify called correctly? No
Enter violated conditions: Arg3≠Actual3
                                                 INTERACTION WINDOW
```

Frame 2 illustrates DEBUSSI querying the user for additional information. In the previous frame, dependency and split analysis significantly reduce the number of suspects. But without additional information DEBUSSI is unable to further localize the bug. By querying the user, DEBUSSI incrementally obtains specification information, which can be applied to the task of reducing the number of suspects.

The suspect for a query must be chosen carefully. DEBUSSI uses heuristics to balance the desire to minimize the amount of user interaction with the desire to obtain as much new information as possible. A useful tactic is to avoid asking about suspects that are not likely to be buggy. For example, `Car` and `Cdr` are Lisp primitives, which can be buggy only by being called improperly. Therefore, in this example, DEBUSSI chooses one of the calls to `Unify` for its query.

DEBUSSI asks if `Unify` was called correctly. The user answers that it was not. By being able to decide whether or not `Unify` was called correctly, the user has revealed that he possesses some bit of extra knowledge about the program. DEBUSSI elicits this information by asking the user to supply a violated condition.

The user supplies the violated condition `Arg3≠Actual3`. As a convenience to the user, DEBUSSI establishes a lexical environment for queries wherein arguments and their values can be referred to by distinguised symbols (e.g., `Arg3` denotes the third argument to `Unify`, and `Actual3` denotes the value supplied as the third argument). Thus the condition supplied by the user says the third argument given to the recursive call to `Unify` should not have been the value which was actually supplied.

This query allows DEBUSSI to further localize the bug. In particular, because neither call to `Cdr` contributes to the incorrect argument to `Unify`, they are both exonerated.

Frame 3

```
Function call: (Unify '((? X)) '((? X)) Nil)
Returned (incorrect) value: ((? X) . (? X))
```

```
(Defun Unify (P1 P2 Env)
  (Cond ((Eq Env 'NoMatch) 'NoMatch)
        ((Var? P1)
         (If (Var? P2)
             (Var-Var-Match P1 P2 Env)
           (Var-Pat-Match P1 P2 Env)))
        ((Var? P2) (Var-Pat-Match P2 P1 Env))
        ((Constant? P1)
         (If (Constant? P2)
             (If (Same-Constant? P1 P2)
                 Env
               'NoMatch)
           'NoMatch))
        ((Constant? P2) 'NoMatch)
        (T (Unify (Cdr P1)
                  (Cdr P2)
                  (Unify (Car P1) (Car P2) Env)))))
                                                          CODE WINDOW
```

```
Dependency analysis exonerates:
  Unify Cdr

Function call: (Unify '(? X) '(? X) Nil)
Returned (incorrect) value: (((? X) . (? X)))

Was Unify called correctly? Yes
A bug is within the call to Unify.  Zooming in...
                                                   INTERACTION WINDOW
```

The boxed call to `Unify` supplies the third argument to the underlined call to `Unify`. Since the previous query indicated that the third argument to underlined call was incorrect, DEBUSSI need not ask the user about the output of the boxed call to `Unify`. DEBUSSI proceeds to ask if the other call to `Unify` was made with the correct arguments, and the user answers that it was. Any call that produces an incorrect output given correct inputs is buggy, so DEBUSSI concludes that bug is within `Unify`. Having localized a bug to a particular function call, DEBUSSI continues by expanding the function call in line ("zooming in"), and attempting to localize the bug to some lower-level function call.

In the process of localizing the bug to this call to `Unify`, DEBUSSI has obtained a new test case for `Unify`, i.e., that `(Unify '(? X) '(? X) Nil)` should produce an output of `Nil`. This is an example of how DEBUSSI incrementally acquires partial specification information.

Frame 3.1 (Not Shown to the User)

```
Function call: (Unify '(? X) '(? X) Nil)
Returned (incorrect) value: (((? X) . (? X)))
```

```
(Defun Unify (P1 P2 Env)
  (Cond ((Eq Env 'NoMatch) 'NoMatch)
        ((Var? P1)
         (If (Var? P2)
             (Var-Var-Match P1 P2 Env)
           (Var-Pat-Match P1 P2 Env)))
        ((Var? P2) (Var-Pat-Match P2 P1 Env))
        ((Constant? P1)
         (If (Constant? P2)
             (If (Same-Constant? P1 P2)
                 Env
               'NoMatch)
           'NoMatch))
        ((Constant? P2) 'NoMatch)
        (T (Unify (Cdr P1)
                  (Cdr P2)
                  (Unify (Car P1) (Car P2) Env)))))
                                                        CODE WINDOW
```

```
Suspects due to dependency analysis:
  Eq Var? Var-Var-Match
Split analysis exonerates:
  Eq Var?

A bug is within the call to Var-Var-Match.  Zooming in...
                                                INTERACTION WINDOW
```

In Frame 3.1, DEBUSSI repeats the process of dependency and split analysis on the next recursive call to **Unify**. Within this call, **Var-Var-Match** is suspected because of data flow dependence. Because the execution of **Var-Var-Match** was determined by the outcomes of **Eq** and **Var?**, they are suspected through control flow dependence.

Split analysis exonerates all of the suspects except **Var-Var-Match**. Thus DEBUSSI concludes that the bug is within **Var-Var-Match**, and continues onward by attempting to localize the bug within **Var-Var-Match**.

Frame 3.2 (Not Shown to the User)

```
Function call: (Var-Var-Match '(? X) '(? X) Nil)
Returned (incorrect) value: (((? X) . (? X)))
```

```
(Defun Var-Var-Match (V1 V2 Env)
  (If (Eq V1 V2)
      Env
    (Let ((B1 (Lookup V1 Env))
          (B2 (Lookup V2 Env)))
      (If (Null B1)
          (If (Null B2)
              (Extend V1 V2 Env)
            (Unify V1 (Binding-Val B2) Env))
        (Unify (Binding-Val B1)
               (If (Null B2)
                   V2
                 (Binding-Val B2))
               Env)))))
                                                        CODE WINDOW
```

```
Suspects due to dependency analysis:
  Eq Lookup Null Extend
Split analysis exonerates:
  Null
Dependency analysis exonerates:
  Lookup
                                                 INTERACTION WINDOW
```

Frame 3.2 again demonstrates DEBUSSI's use of dependency analysis and split analysis, with split analysis exonerating both calls to Null. As an indirect result of Null's exoneration, the two calls to Lookup are also exonerated. For example, the first call to Null is the sole consumer of the output produced by the first call to Lookup. When Null is exonerated, the data produced by Lookup no longer contributes to the bug manifestation, so Lookup may also be exonerated.

Frame 4

```
Function call: (Var-Var-Match '(? X) '(? X) Nil)
Returned (incorrect) value: (((? X) . (? X)))
```

```
(Defun Var-Var-Match (V1 V2 Env)
  (If (Eq V1 V2)
      Env
    (Let ((B1 (Lookup V1 Env))
          (B2 (Lookup V2 Env)))
      (If (Null B1)
          (If (Null B2)
              (Extend V1 V2 Env)
            (Unify V1 (Binding-Val B2) Env))
        (Unify (Binding-Val B1)
               (If (Null B2)
                   V2
                 (Binding-Val B2))
               Env)))))
                                                          CODE WINDOW
```

```
Function call: (Extend '(? X) '(? X) Nil)
Returned (incorrect) value: (((? X) . (? X)))

Was Extend called correctly? No
Enter violated conditions: ¬Executed(Extend)

Dependency analysis exonerates:
  Extend

A bug is within the call to Eq.
Returned (incorrect) value: Nil
                                                   INTERACTION WINDOW
```

The bug is localized in Frame 4. DEBUSSI knows that **Extend** returned an incorrect value from the test case. It does not know, however, whether **Extend** was called correctly. The user responds that **Extend** was not called correctly because, in this test case, **Extend** should not have been called at all. This fact is stated in the violated condition **¬Executed(Extend)**. This final query leaves **Eq** as the sole suspect, and DEBUSSI thereby concludes that the bug must lie in **Eq**.

The bug with **Eq** is attributable to the way Common Lisp tests for equality. **Eq** returns **T** only when the two items being compared point to the same location in memory. In **Var-Var-Match**, the items in question are lists that represent pattern variables. Even if these lists appear the same when printed, there is no guarantee that they refer to the same memory location. Thus **Eq** is the wrong predicate for comparing two variables.

The predicate which should have been used in `Var-Var-Match` is `Equal`, which returns `T` when the two items being compared have the same printed representation. This means that when two variables print the same, they *are* the same for the purposes of matching.

This type of bug is quite common in Lisp programming. A practical implementation of `Unify` would almost certainly employ a "read macro" to simplify the typing of variables. Specifically, the Lisp reader would be customized to make tokens like `?X` expand into `(? X)`. A programmer who didn't know about this customization would perceive variables such as `?X` to be Lisp "atoms," and would therefore see no fault in comparing two variables via `Eq`.

2.5 Localizing a Fatal Bug

Recall that there are two bugs in the implementation of `Unify`. The first bug involved `Eq`, and manifested itself by producing an incorrect output. After fixing the first bug, the user tests `Unify` with a different test case.

Frame 5

```
(Defun Unify (P1 P2 Env)
  (Cond ((Eq Env 'NoMatch) 'NoMatch)
        ((Var? P1)
         (If (Var? P2)
             (Var-Var-Match P1 P2 Env)
           (Var-Pat-Match P1 P2 Env)))
        ((Var? P2) (Var-Pat-Match P2 P1 Env))
        ((Constant? P1)
         (If (Constant? P2)
             (If (Same-Constant? P1 P2)
                 Env
               'NoMatch)
           'NoMatch))
        ((Constant? P2) 'NoMatch)
        (T (Unify (Cdr P1)
                  (Cdr P2)
                  (Unify (Car P1) (Car P2) Env)))))
                                                          CODE WINDOW
```

```
> (Unify '(? X) '(? Y) '(((? X) . 1)))

Trap: The first argument given to the CAR instruction, ?,
      was not a locative, a cons, or NIL.

→ Show Backtrace
Binding-Value ← Var-Var-Match ← Unify

→ (Correct-Output-Is '(((? X) . 1) ((? Y) . 1)))
                                                   INTERACTION WINDOW
```

The user tests `Unify`, resulting in a run-time error. Examining the stack indicates that the error occured within a call to `Binding-Value`. It is incorrect to assume that the bug lies in some function call on the stack. For example, the program may have bombed because some previously executed function supplied `Binding-Value` with an incorrect argument. Or it may have bombed because some control-flow split caused `Binding-Value` to be executed unexpectedly. In these situations the culprit has already completed its execution, and will therefore not appear on the runtime stack.

What appears on the runtime stack is of little importance to DEBUSSI's localization strategies. DEBUSSI looks for suspects by analyzing the dependencies that lead to a bug manifestation. If `Binding-Value` was indeed supplied with a bad argument, then this fact will show up in the underlying dependency structure, not on the stack.

Just as in the previous example, the user invokes DEBUSSI by specifying the expected program output with a call to `Correct-Output-Is`. Note that the Lisp runtime environment has been enhanced to allow DEBUSSI to be invoked in the midst of a runtime error.

Frame 5.1 (Not Shown to the User)

```
Function call: (Unify '(? X) '(? Y) '(((? X) . 1)))
A run-time error occurred in Unify.
```

```
(Defun Unify (P1 P2 Env)
  (Cond ((Eq Env 'NoMatch) 'NoMatch)
        ((Var? P1)
         (If (Var? P2)
             (Var-Var-Match P1 P2 Env)
           (Var-Pat-Match P1 P2 Env)))
        ((Var? P2) (Var-Pat-Match P2 P1 Env))
        ((Constant? P1)
         (If (Constant? P2)
             (If (Same-Constant? P1 P2)
                 Env
               'NoMatch)
           'NoMatch))
        ((Constant? P2) 'NoMatch)
        (T (Unify (Cdr P1)
                  (Cdr P2)
                  (Unify (Car P1) (Car P2) Env)))))
                                                        CODE WINDOW
```

```
Suspects due to dependency analysis:
  Eq Var? Var-Var-Match
Split analysis exonerates:
  Eq Var?

A bug is within the call to Var-Var-Match.  Zooming in...
                                                  INTERACTION WINDOW
```

Frame 5.1 indicates that Var-Var-Match is the only suspect to withstand dependency analysis and split analysis. DEBUSSI therefore concludes that the bug must be within it. As previously stated, the fact that Var-Var-Match is the next function on the runtime stack is merely coincidental. DEBUSSI focuses on Var-Var-Match as a result of its reasoning about dependencies.

Frame 6

```
Function call: (Var-Var-Match '(? X) '(? Y) '(((? X) . 1)))
A run-time error occurred in Var-Var-Match.
```

```
(Defun Var-Var-Match (V1 V2 Env)
  (If (Equal V1 V2)
      Env
    (Let ((B1 (Lookup V1 Env))
          (B2 (Lookup V2 Env)))
      (If (Null B1)
          (If (Null B2)
              (Extend V1 V2 Env)
            (Unify V1 (Binding-Val B2) Env))
        (Unify (Binding-Val B1)
               (If (Null B2)
                   V2
                 (Binding-Val B2))
               Env)))))
                                                      CODE WINDOW
```

```
Suspects due to dependency analysis:
  Equal Lookup Null Binding-Val
Split analysis exonerates:
  Equal Null

Function call: (Lookup (? X) (((? X) . 1)))
Returned value: ?
Did Lookup return the right result? No

A bug is within the call to Lookup.  Zooming in...
                                                 INTERACTION WINDOW
```

In Frame 6, the first call to Lookup is suspected, while the second call is not. This happens because in this particular execution, there were no consumers for the second call's output. In other words, the bug in **Var-Var-Match** did not depend on the second call to Lookup. Since Lookup is a user-written function, DEBUSSI chooses it for a query.

The first call to Lookup is already known to be called correctly by virtue of the test case. However, the user indicates that the value returned by Lookup is not correct. Thus the bug is further localized, and DEBUSSI focuses on Lookup.

Frame 7

```
Function call: (Lookup '(? X) '(((? X) . 1)))
Returned (incorrect) value: ?
```

```
(Defun Lookup (Var Env)
  (Cond ((Null Env) Nil)
        ((Equal Var (Binding-Var (Car Env))) (Car Var))
        (T (Lookup Var (Cdr Env)))))
                                                    CODE WINDOW
```

```
Suspects due to dependency analysis:
  Null Equal Car Binding-Var
Split analysis exonerates:
  Null Equal
Dependency analysis exonerates:
  Car Binding-Var

A bug is within the call to Car.
Returned (incorrect) value: ?
                                                INTERACTION WINDOW
```

Dependency and split analysis rule out all suspects except Binding-Var and the two calls to Car. But the only function call that depends on Binding-Var and the first call on Car is Equal. Since Equal has been exonerated, Binding-Var and Car can be exonerated as well. This leaves only one suspect, the second call to Car.

Car is buggy because it was called with the wrong argument. Within Lookup, Car should have returned (Car Env) instead of (Car Var). The purpose in calling (Car Env) is to return the binding pair associated with a given variable, so that callers of Lookup can use Binding-Var and Binding-Val to examine its result. Calling (Car Var) merely returns a question mark "?," which is obviously not the intent of Lookup.

Note that the error in this function is in its data flow, and not in its use of Car. Reparing this bug involves replacing the Var argument to Car with Env. As will be discussed in Chapter 3, DEBUSSI is not particularly adept at localizing errors in data flow. In this particular example, however, the bug was localized because DEBUSSI was able to find a language primitive that returned an incorrect result given correct arguments.

2.6 A Partially Implemented Program

This final scenario demonstrates how DEBUSSI localizes bugs in a partially implemented program. The process of debugging partially implemented programs highlights DEBUSSI's use of automated reasoning in the symbolic evaluation of specifications. Furthermore, the scenario illustrates the advantages gained by the ability to debug programs in early stages of their design. The ability to debug partially implemented programs is due to DEBUSSI's dependency directed approach.

The example used in this scenario is a program that computes (at the level of a video game) the estimated time to arrival (ETA) of a "space ship" to an "asteroid." ETA is defined to mean the finite, positive amount of time that will elapse before a space ship arrives at an asteroid. If the space ship will never arrive, or if it has already arrived, then the ETA is undefined.

The space ship is represented as a point in space $\vec{x}_s$ moving in a straight line at a constant velocity $\vec{v}_s$, via the equation

$$\vec{x}_s = \vec{x}_{s_0} + \vec{v}_s t.$$

The asteroid is represented as a circle in space centered at $\vec{x}_a$ and having radius r. Points $\vec{x}$ on the surface of the asteroid will satisfy the equation

$$\|\vec{x} - \vec{x}_a\|^2 = r^2.$$

Just like the space ship, the asteroid moves in a straight line at a constant velocity $\vec{v}_a$,

$$\vec{x}_a = \vec{x}_{a_0} + \vec{v}_a t.$$

The problem is made simpler by shifting the frame of reference to make the asteroid stationary at the origin. After this change in reference, the surface of the asteroid is described by

$$\|\vec{x}\|^2 = r^2$$

and the motion of the space ship is described by

$$\vec{x}'_s = \vec{x}_0 + \vec{v}t,$$

where

$$\vec{x}_0 = \vec{x}_{s_0} - \vec{x}_{a_0}, \text{ and}$$
$$\vec{v} = \vec{v}_s - \vec{v}_a.$$

Determining the ETA between the space ship and asteroid requires determining value(s) of t where the trajectory of the space ship intersects the surface of the asteroid. These values can be found by substituting the equation describing the motion of the ship into the equation describing the surface of the asteroid

$$\|\vec{x}'_s\|^2 = r^2$$

Substituting for $\vec{x}'_s$ and rearranging terms yields the following quadratic in t, which we call the "intersection equation."

$$\|\vec{v}\|^2 t^2 + 2\vec{x}_0 \cdot \vec{v}\, t + \|\vec{x}_0\|^2 - r^2 = 0$$

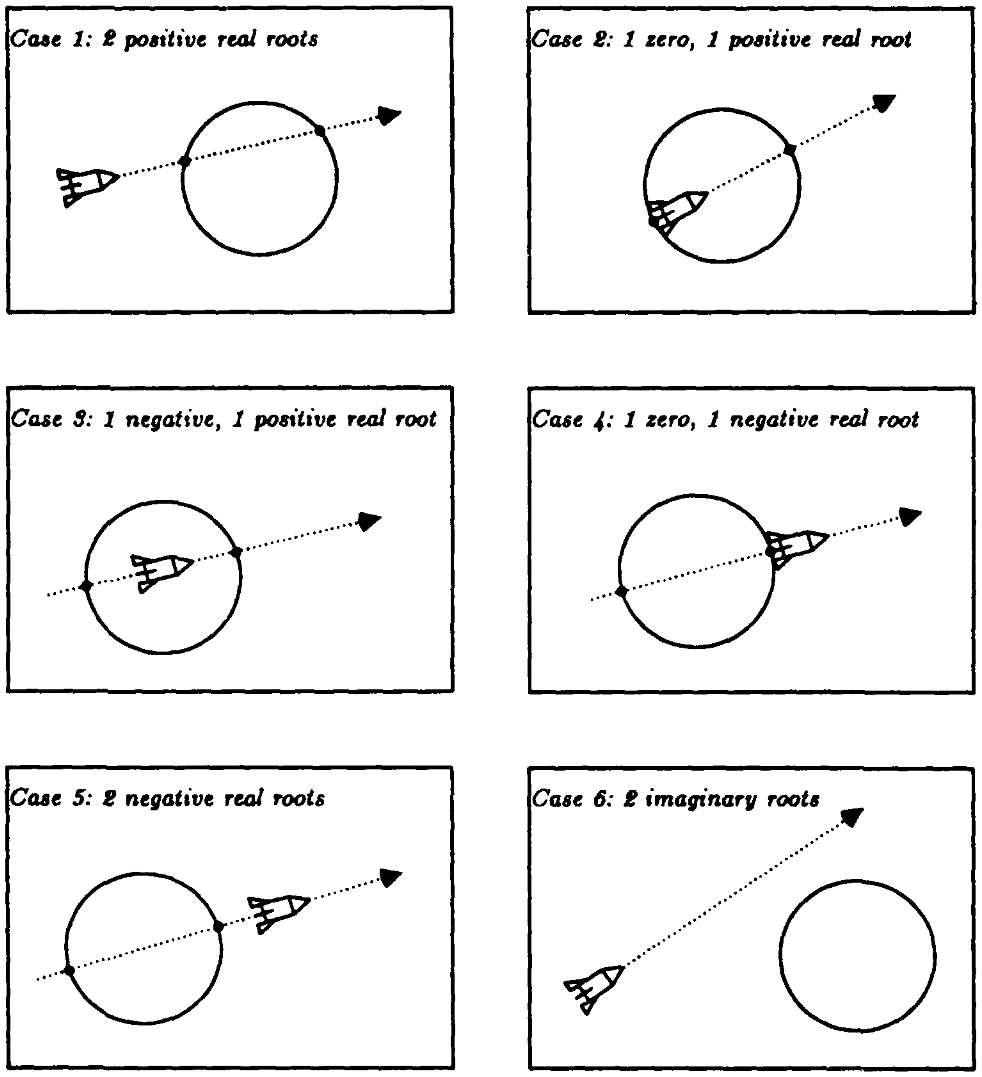

Case 1: 2 positive real roots
Case 2: 1 zero, 1 positive real root
Case 3: 1 negative, 1 positive real root
Case 4: 1 zero, 1 negative real root
Case 5: 2 negative real roots
Case 6: 2 imaginary roots

Figure 2.3: Interpretation of solutions to the intersection equation

Solving the intersection equation yields two complex time values, which must be correctly interpreted in order to find the true ETA. Figure 2.3 depicts how the initial position and velocity of the space ship determine the two solutions to the intersection equation. Most of the cases in the figure are situations where the ETA is undefined. Cases 2 and 3 place the space ship at a physically implausible location within the asteroid; these cases will never be encountered in the normal execution of the program. In cases 4 and 5 the space ship has already arrived at the asteroid. Finally, in case 6, the space ship misses the asteroid entirely.

```
Type Ship-Data
  Parts Origin: Vector,
        Velocity: Vector

Type Asteroid-Data
  Parts Origin: Vector,
        Velocity: Vector,
        Radius: Number

IOSpec ETA-To-Asteroid
  Inputs Ship: Ship-Data, Asteroid: Asteroid-Data
  Outputs ETA: Real-Or-Undefined
  Postconditions
    Real(ETA) → ETA > 0

IOSpec Quadratic-Roots
  Inputs A: Number, B: Number, C: Number
  Outputs Root1: Complex, Root2: Complex
  Postconditions
    Root1 = (-B + Sqrt(B^2 - 4*A*C)) / (2*A)
    Root2 = (-B - Sqrt(B^2 - 4*A*C)) / (2*A)
```

Figure 2.4: Programmer-Defined Partial Specifications for ETA-To-Asteroid

The only situation that will yield a valid ETA is shown in case 1. In this case, the ETA is the smaller of the two positive real roots. This yields the general rule for finding the ETA: *The* ETA *is the smallest positive real root of the intersection equation.*

Figure 2.4 shows the partial specifications (written by the programmer) for the program that will compute the ETA. The first two specifications define data types that will be used in the program. **Ship-Data** describes a space ship in terms of its origin and velocity, and **Asteroid-Data** describes an asteroid in terms of its origin, velocity and radius.

The third specification in the Figure is for **ETA-To-Asteroid**, the top-level function that will determine the ETA. This specification is partial. It makes no mention of how the ETA is computed, only stating that if the ETA is real, it must be greater than zero.

The implementation of **ETA-To-Asteroid** will require a function to find the roots of a quadratic equation. Rather than write the function right away, the programmer supplies a specification for it. This specification, **Quadratic-Roots**, is the fourth I/O specification shown in Figure 2.4.

Figure 2.5 describes other data types and functions that will be used in the im-

```
Type Number
  Includes Real, Integer, Imaginary

Type Complex
  Parts Re: Real, Im: Imaginary;

Type Vector

Function Vector+ (X: Vector, Y: Vector) : Vector

Function Vector- (X: Vector, Y: Vector) : Vector

Function Vector-Dot (X: Vector, Y: Vector) : Number

Function Vector-Scale (N: Number, X: Vector) : Vector
```

Figure 2.5: Predefined auxiliary specifications

```
(Defstruct (Ship)
  Origin Velocity)

(Defstruct (Asteroid)
  Origin Velocity Radius)

(Defun ETA-To-Asteroid (Ship Asteroid)
  (Let* ((Origin (Vector- (Origin Ship) (Origin Asteroid)))
         (Velocity (Vector- (Velocity Ship) (Velocity Asteroid)))
         (A (Vector-Dot Velocity Velocity))
         (B (* 2 (Vector-Dot Origin Velocity))`
         (C (- (Vector-Dot Origin Origin)
               (* (Radius Asteroid) (Radius Asteroid)))))
    (Multiple-Value-Bind (Root1 Root2)
        (Quadratic-Roots A B C)
      (Min (Abs Root1) (Abs Root2)))))
```

Figure 2.6: Implementation of ETA-To-Asteroid

plementation of `ETA-To-Asteroid`. They are assumed to be part of some predefined library that is available to the programmer. In the figure, `Number` denotes any real, integer or imaginary number, `Complex` represents a complex number consisting of real and imaginary parts, and `Vector` represents a vector of unspecified dimension.

The final functions in Figure 2.5 perform vector operations. `Vector+` and `Vector-` perform vector addition and subtraction. `Vector-Dot` performs a vector dot-product, and `Vector-Scale` multiplies a vector by a scalar.

The implementation of `ETA-To-Asteroid` is shown in Figure 2.6. (By convention, the name of the function which implements an I/O specification will have the same name as the specification.) Some of the functions used in this program, such as `Quadratic-Roots`, are not yet implemented, and exist only as specifications. Other functions, such as `Vector+`, are implemented (in the library) but details about their specification and implementation are not presented here.

The last line of `ETA-To-Asteroid` is meant by programmer to determine the smallest positive root found by `Quadratic-Roots`. It is in fact a naive and incorrect implementation, because it disregards the possibility of a root being negative or imaginary. DEBUSSI is able to localize this bug despite the incompleteness and abstractness of the program. This illustrates the utility of dependency-directed localization for debugging programs in the early stages of their development.

Frame 8

```
(Defun ETA-To-Asteroid (Ship Asteroid)
  (Let* ((Origin (Vector- (Origin Ship) (Origin Asteroid)))
         (Velocity (Vector- (Velocity Ship) (Velocity Asteroid)))
         (A (Vector-Dot Velocity Velocity))
         (B (* 2 (Vector-Dot Origin Velocity)))
         (C (- (Vector-Dot Origin Origin)
               (* (Radius Asteroid) (Radius Asteroid)))))
    (Multiple-Value-Bind (Root1 Root2)
        (Quadratic-Roots A B C)
      (Min (Abs Root1) (Abs Root2)))))
                                                         CODE WINDOW
```

```
> (Symbolic-Mode)
:: Declare N: Real
:: Declare ROID: Asteroid-Data
:: Origin(ROID) = [0]
:: Velocity(ROID) = [N]
:: Radius(ROID) = N
:: Declare VOYAGER: Ship-Data
:: Origin(VOYAGER) = [N]
:: Velocity(VOYAGER) = [N]
                                                  INTERACTION WINDOW
```

In Frame 8, the user is testing ETA-To-Asteroid. Since the program is only partially implemented, the user cannot test it with concrete data. Instead, he chooses a test case that deals with underspecified and abstract data.

In order to describe the abstract test case, the user needs to consult with the reasoning system. This is initiated by the call to Symbolic-Mode, to which the system responds by changing the prompt to a double-colon "::".

The user defines the abstract test data as follows. N is an arbitrary real number. ROID is an asteroid at the origin with radius equal to N, and velocity equal to the one-dimensional vector [N]. (Vectors are denoted by square brackets.) VOYAGER is a space ship with origin and velocity equal to [N]. This data establishes the initial conditions leading to Case 4 in Figure 2.3.

Frame 9

```
Function call: (ETA-To-Asteroid VOYAGER ROID)
A specification violation occurred within ETA-To-Asteroid
```

```
(Defun ETA-To-Asteroid (Ship Asteroid)
  (Let* ((Origin (Vector- (Origin Ship) (Origin Asteroid)))
         (Velocity (Vector- (Velocity Ship) (Velocity Asteroid)))
         (A (Vector-Dot Velocity Velocity))
         (B (* 2 (Vector-Dot Origin Velocity)))
         (C (- (Vector-Dot Origin Origin)
               (* (Radius Asteroid) (Radius Asteroid)))))
    (Multiple-Value-Bind (Root1 Root2)
        (Quadratic-Roots A B C)
      (Min (Abs Root1) (Abs Root2)))))
                                                    CODE WINDOW
```

```
:: Run (ETA-To-Asteroid VOYAGER ROID)

Specification violation:
  (ETA-To-Asteroid VOYAGER ROID) = 0
is in conflict with postcondition of ETA-To-Asteroid
  Real(ETA) → ETA > 0

Supects due to dependency analysis:
  Vector- Vector-Dot - * Min Abs Quadratic-Roots
                                              INTERACTION WINDOW
```

The user types "Run (ETA-To-Asteroid VOYAGER ROID)," asking the reasoning system to make whatever deductions it can based on its available specification information. These deductions will involve abstract algebraic reasoning, such as recognizing identities like X+0=X. Abstract reasoning allows DEBUSSI to simulate the execution of the partially implemented program in the absence of explicit numerical values.

A bug manifestation arises as a conflict between the value produced by the simulation of ETA-To-Asteroid and its partial specification. Simulation produces a result of zero, yet the specification dictates that the result be greater than zero.

DEBUSSI depends on an automated reasoning system to perform all deductions. This reasoning system records dependencies that explain how one deduction follows from another. These dependencies are analyzed by DEBUSSI to find an initial set of suspects.

Frame 10

```
Function call: (ETA-To-Asteroid VOYAGER ROID)
A specification violation occurred within ETA-To-Asteroid
```

```
(Defun ETA-To-Asteroid (Ship Asteroid)
  (Let* ((Origin (Vector- (Origin Ship) (Origin Asteroid)))
         (Velocity (Vector- (Velocity Ship) (Velocity Asteroid)))
         (A (Vector-Dot Velocity Velocity))
         (B (* 2 (Vector-Dot Origin Velocity)))
         (C (- (Vector-Dot Origin Origin)
               (* (Radius Asteroid) (Radius Asteroid)))))
    (Multiple-Value-Bind (Root1 Root2)
        (Quadratic-Roots A B C)
      (Min (Abs Root1) (Abs Root2)))))
                                                      CODE WINDOW
```

```
Function call: (Quadratic-Roots (* N N) (* 2 (* N N)) 0)
Returned values: 0 -2

Was Quadratic-Roots called correctly? Yes
Did Quadratic-Roots return the right result? Yes

Dependency analysis exonerates:
  Quadratic-Roots Vector- Vector-Dot - *
                                                 INTERACTION WINDOW
```

Quadratic-Roots is the only user-defined function among the suspects, so it is the first choice for a query. This query reveals that DEBUSSI is mixing reasoning between symbolic and literal data, because the arguments to Quadratic-Roots are presented as symbolic values such as (* N N), and the returned values are presented as actual numbers.

The following illustrates the kind of algebraic simplification used by DEBUSSI in determining the first solution to Quadratic-Roots

```
Root1 = (-(2*N*N) + Sqrt((2*N*N)^2 - 4*(N*N)*0)) / (2*(N*N))
      = (-(2*N*N) + Sqrt((2*N*N)^2 - 0)        / (2*(N*N))
      = (-(2*N*N) + Sqrt((2*N*N)^2))           / (2*(N*N))
      = (-(2*N*N) +      (2*N*N))              / (2*(N*N))
      = 0                                      / (2*(N*N))
Root1 = 0
```

Frame 11

```
Function call: (ETA-To-Asteroid VOYAGER ROID)
A specification violation occurred within ETA-To-Asteroid
```

```
(Defun ETA-To-Asteroid (Ship Asteroid)
  (Let* ((Origin (Vector- (Origin Ship) (Origin Asteroid)))
         (Velocity (Vector- (Velocity Ship) (Velocity Asteroid)))
         (A (Vector-Dot Velocity Velocity))
         (B (* 2 (Vector-Dot Origin Velocity)))
         (C (- (Vector-Dot Origin Origin)
               (* (Radius Asteroid) (Radius Asteroid)))))
    (Multiple-Value-Bind (Root1 Root2)
        (Quadratic-Roots A B C)
      (Min (Abs Root1) (Abs Root2)))))
                                                        CODE WINDOW
```

```
Function call: (Min 0 2)
Returned (incorrect) value: 0

Was Min called correctly? No
Enter violated conditions: Arg1≠0

Dependency analysis exonerates:
  Min Abs

A bug is within the call to Abs.
Returned (incorrect) value: 0
                                                 INTERACTION WINDOW
```

In Frame 11, a query about a call to Min leads to the localization of the bug to a call to Abs. Seeing the call to Min makes the user realize that he oversimplifed the task of finding the smallest positive root. Allowing Min to be called with a zero argument leads to the possiblity of ETA-To-Asteroid returning zero, which would violate the specification that the ETA be positive. As a result of this new information, DEBUSSI can localize the bug to the call to Abs.

Localizing the bug to Abs illustrates how DEBUSSI deals with multiple bugs. The problem in ETA-To-Asteroid actually involves both of the calls to Abs and the call to Min. Rather than attempting to prove that these three function calls conspire to violate ETA-To-Asteroid's specification, DEBUSSI adopts a "one bug at a time" strategy, finding one function call that contributes to the bug manifestation.

Frame 12

```
(Defun ETA-To-Asteroid (Ship Asteroid)
  (Let* ((Origin (Vector- (Origin Ship) (Origin Asteroid)))
         (Velocity (Vector- (Velocity Ship) (Velocity Asteroid)))
         (A (Vector-Dot Velocity Velocity))
         (B (* 2 (Vector-Dot Origin Velocity)))
         (C (- (Vector-Dot Origin Origin)
               (* (Radius Asteroid) (Radius Asteroid)))))
    (Multiple-Value-Bind (Root1 Root2)
        (Quadratic-Roots A B C)
      (WHEN (AND (REALP ROOT1) (REALP ROOT2))
        (COND
          ((AND (PLUSP ROOT1) (PLUSP ROOT2)) (MIN ROOT1 ROOT2))
          ((PLUSP ROOT1) ROOT1)
          ((PLUSP ROOT2) ROOT2))))))
                                                            CODE WINDOW
```

```
Finished editing ETA-To-Asteroid.

:: Run (ETA-To-Asteroid VOYAGER ROID)
(ETA-To-Asteroid VOYAGER ROID) = Nil
                                                     INTERACTION WINDOW
```

In Frame 12, the user has finished correcting the bug. (The new code is shown in the code window in a slanted font.) Retesting `ETA-To-Asteroid` with the original test case now produces the expected answer, `Nil`. Since the postcondition of `ETA-To-Asteroid` only describes its behavior in the case where its result is a number, returning `Nil` does not violate its partial specification.

Chapter 3

The Details

The previous chapter presented DEBUSSI as it appears from the user's perspective. From this point of view, DEBUSSI operates on programs as source code, and bug manifestations are incorrect concrete or symbolic values. To the user, bug localization involves reasoning about the variables and control constructs in source code.

DEBUSSI's bug localization techniques are much more general, however. Within DEBUSSI, programs are represented in the Plan Calculus [2, 13, 16]. Bug manifestations are contradictions in the reasoning system that arise when specifications are violated. Bug localization involves *reasoning about* the dependencies that lead to a contradiction.

The same techniques used by DEBUSSI to debug fully implemented programs can also be applied to partially implemented programs. Similarly, the techniques that allow reasoning about concrete data (e.g., $Foo = 3$) also allow reasoning about more abstract specifications (e.g., $Foo > 0$).

An important part of DEBUSSI's bug localization methodology is its query mechanism. Queries allow DEBUSSI to incrementally acquire partial specification information. DEBUSSI uses this new information to narrow down the set of suspects as well as to increase its knowledge of the buggy program. An ideal query is one which is easily answered by the user, results in the immediate conviction of a suspect, and gathers significant new specification information. DEBUSSI uses a variety of heuristics in an attempt to approximate this ideal.

This chapter first describes the Plan Calculus, the representation for programs used by DEBUSSI. A diagrammatic notation for the Plan Calculus is used as an aid in visualizing DEBUSSI's dependency-directed reasoning techniques.

The bulk of the chapter demonstrates and generalizes DEBUSSI's localization techniques. The end result is a domain-independent view of bug localization. In domain-independent debugging, bugs are localized solely by reasoning about dependencies—the dependencies are supplied by the domain.

Next, the chapter gives a brief overview of DEBUSSI's localization algorithm. An important part of this discussion is how the algorithm deals with multiple bugs.

The chapter concludes with a discussion of the various heuristics that comprise DEBUSSI's query mechanism.

3.1 The Plan Calculus

DEBUSSI does not operate on programs as source code. It uses the Plan Calculus [13, 16], in which programs are modeled as *plans*. A plan consists of a structural component, shown via a *plan diagram*, and a logical component, consisting of annotations and constraints on the plan diagram.

Plan diagrams are primarily used as tools for visualizing a program's structure and illustrating dependency-directed reasoning. The implementation of the Plan Calculus in Cake that is used by DEBUSSI does not represent plan diagrams directly. Instead, it uses logical predicates to express the information conveyed via plan diagrams.

There are several advantages in using the Plan Calculus to represent programs. First, the Plan Calculus abstracts away from syntactic details about a program, making DEBUSSI essentially programming language independent. The Plan Calculus separates the meaningful operations performed by a program from "connective tissue," such as variable binding constructs. Second, the Plan Calculus is a wide-spectrum representation, allowing DEBUSSI to operate on partially implemented programs (mixtures of specifications and implemented code).

Figure 3.1 shows the plan diagram for `Unify` (from the scenario in Chapter 2). Solid arrows in the diagram represent data flow. Hatched arrows represent control flow. The data flow arcs entering the plan diagram from the top correspond to the arguments to `Unify`. The data flow arc exiting the plan from the bottom corresponds to the result produced by `Unify`.

Boxes in the diagram represent computation steps. Some boxes, such as `Car` and `Var-Var-Match`, are instances of *I/O specifications*, corresponding to simple function calls. Other boxes, such as `Variable?` and `Constant?`, are instances of *split specifications*, corresponding to functions called in conditionals. Note that each split has two control flow outputs labeled `T` and `F`, corresponding to the success and failure of the split's condition. Finally, `Join` boxes represent the recombination of control flow after a split. Joins are a part of the program's data flow and control flow structure.

Reasoning about plans in DEBUSSI is done using Cake [15], a hybrid knowledge representation and reasoning system developed for the Programmer's Apprentice. Cake has two features that make it especially appropriate to debugging. First, it includes a truth maintenance system, allowing it to manage the changing and potentially contradictory information inherent in buggy programs. Second, it supports reasoning with a mixture of concrete and abstract information.

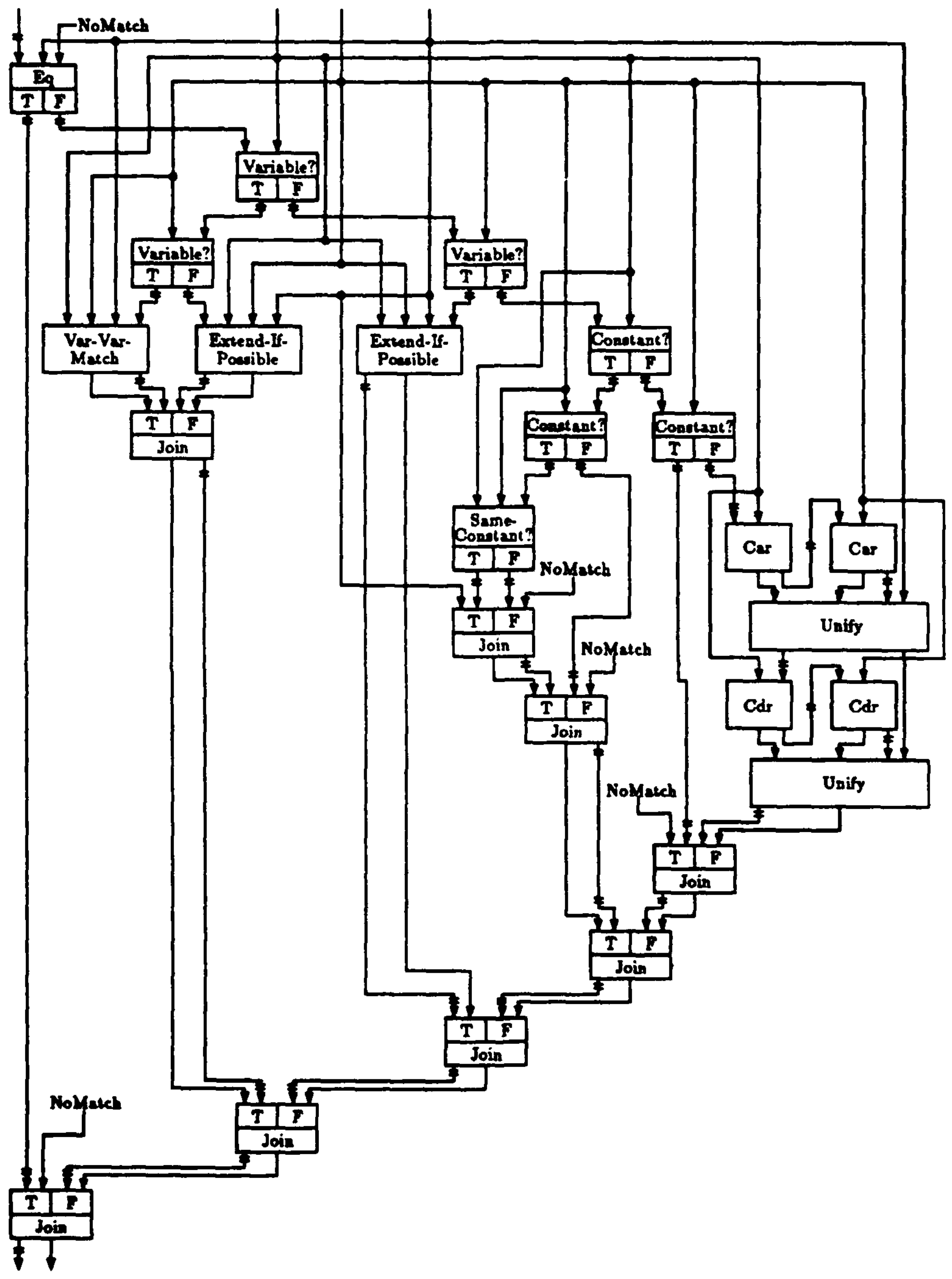


Figure 3.1: Plan diagram for `Unify`

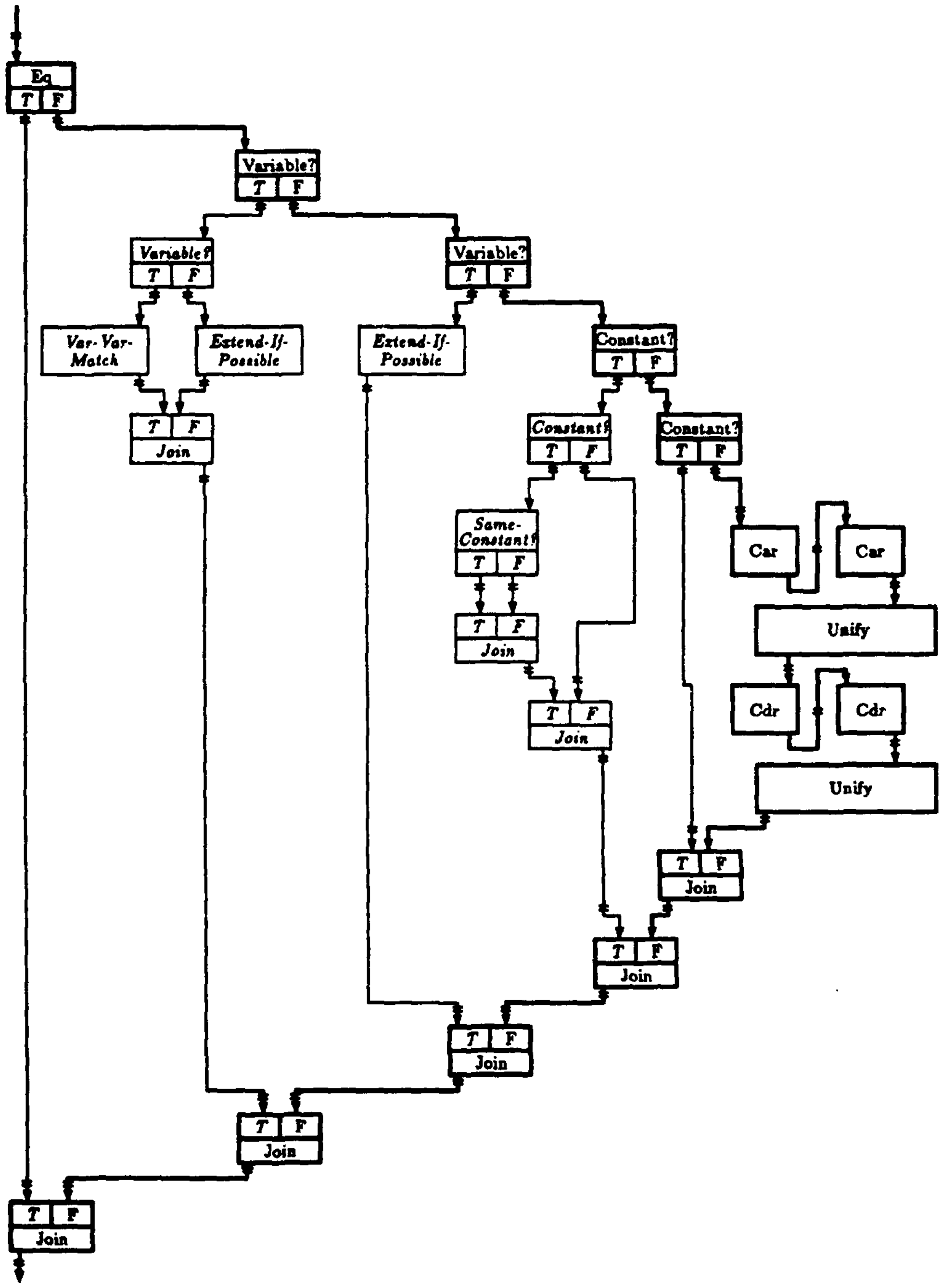


Figure 3.2: Control flow dependency for `Unify` from test case in Frame 2

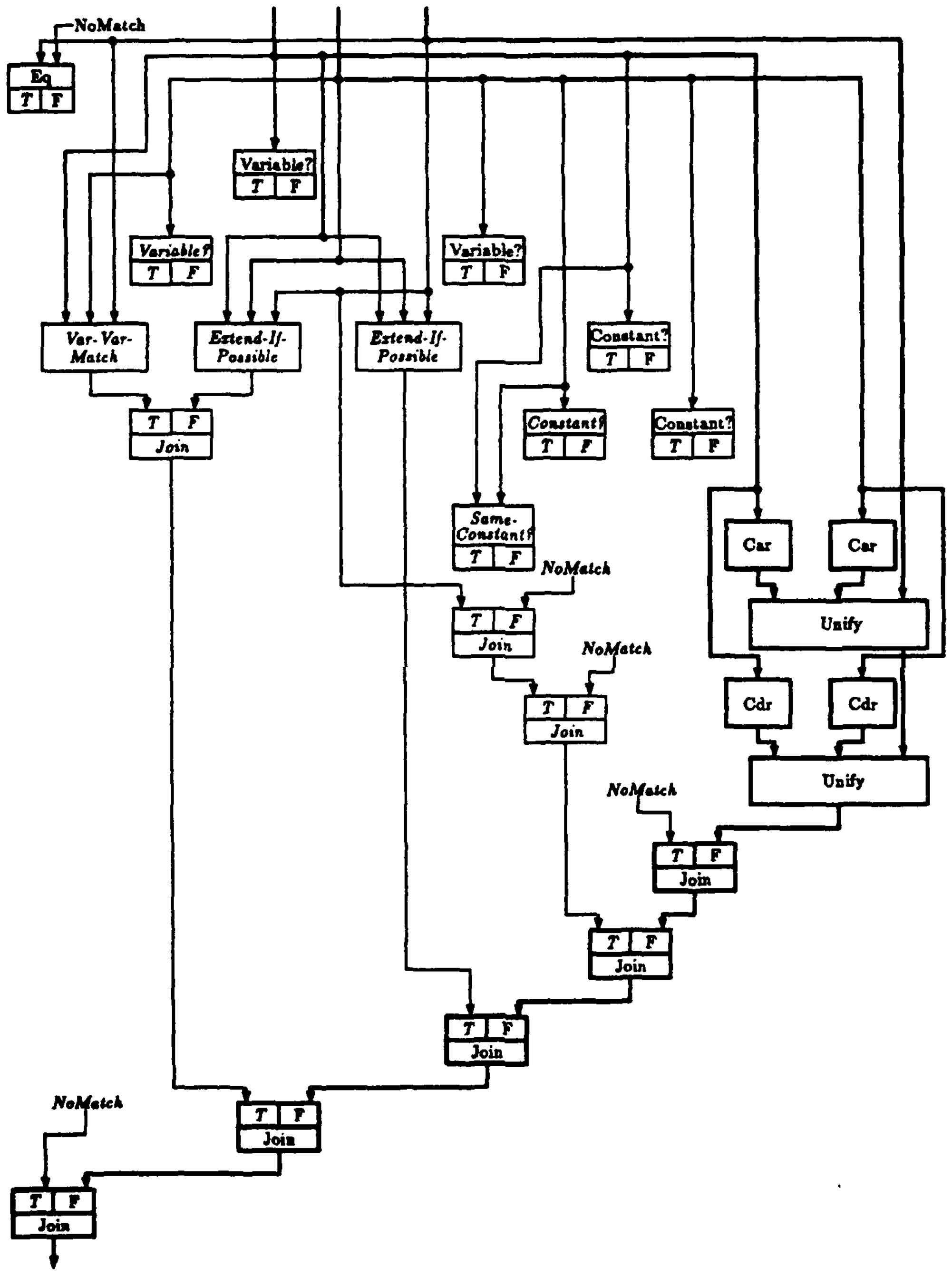


Figure 3.3: Data flow dependency for `Unify` from test case in Frame 2

3.2 Visualizing Dependencies

To illustrate some of the mechanisms that underlie DEBUSSI, we now show how the techniques presented in the scenarios operate on plan diagrams. This section demonstrates dependency tracing as it operates on the plan diagram for `Unify`. (This task was done in Frame 2 of the scenario.)

In Figures 3.2 and 3.3, dependency analysis is visualized as tracing "wires" from the output of a program "upstream" towards the inputs. In the figures, the result of this dependency tracing is indicated via darkened arcs and boxes. Thus every box that is darkened in these figures is a suspect due to dependency analysis.

Figure 3.2 illustrates control flow dependency. A function `F` has control flow dependence on a function `G` if the outcome of `G` determines whether or not `F` is to be executed. For example, in the figure, the first call on `Car` depends on `Constant?` because `Car` was executed as a direct result of the outcome of `Constant?`. Control flow dependence is transitive, e.g., `Car` also has control flow dependence on `Eq`.

Figure 3.3 illustrates data flow dependency. A function `F` has data flow dependence on a function `G` if `F` consumes data produced by `G`. In the figure, the first instance of `Unify` depends on the first call to `Car` because it directly consumes data produced by `Car`. Data flow dependence is also transitive, e.g., the second instance of `Unify` also depends on `Car`.

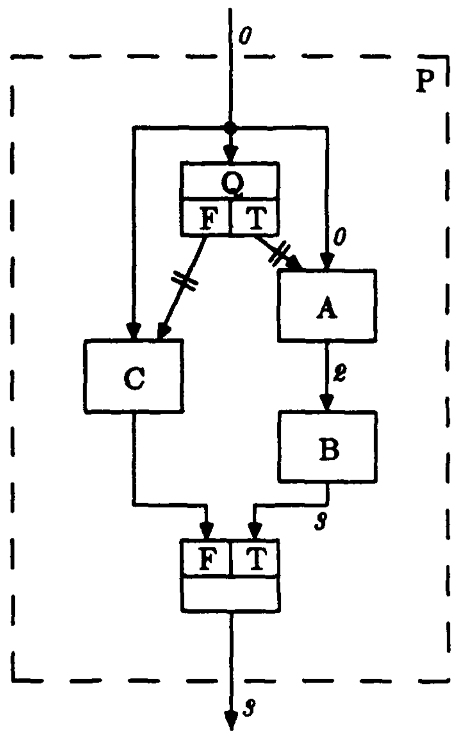
3.3 Dependency-Directed Bug Localization

Bug localization is not a task unique to programming. DEBUSSI's bug localization techniques do not assume that the artifact being debugged is a program. DEBUSSI's view is that a bug is a contradiction in the reasoning system, and a suspect is a premise that underlies that contradiction. The goal of bug localization is to find the smallest set of premises that lead to the contradiction.

This section illustrates the generality of DEBUSSI's localization techniques. The illustrative example will be a program, but the reasoning techniques do not explicitly rely on data and control flow to establish dependencies. Earlier chapters of this report demonstrate the effects of various localization strategies at the source code level. This section gets to the heart of these strategies by demonstrating how they actually operate on dependency structures in the reasoning system.

Figure 3.4 depicts a simple plan diagram and the program it represents. This plan is the basis for this section's discussion of dependency-directed bug localization.

Small numbers next to the data flow arcs in the plan diagram indicate values produced by the program at run time. Reading the figure, we see that the program was tested on an initial input of zero, and produced 3 as a result. This is in conflict with the program's partial specification, which says that the program should produce an output of 2 for an input of zero.



```
(Defun P (x)
  (If (Q x)
      (B (A x))
    (C x)))
```

```
IOSpec P
  Inputs X: Any
  Outputs Result: Any
  Postconditions
    X=0 -> Result=2
```

Figure 3.4: A simple Lisp program, partial specification and plan

The violated program specification leads to a contradiction in the reasoning system. The contradiction and its underlying dependency structure is shown in Figure 3.5. A node in the graph represents a deduction made by the reasoning system; the children of a node represent the logical support for their parent. For example, the deductions $Output(P) = 3$ and $Output(P) = 2$ lead to the contradiction at the root (top) of the dependency graph.

In the figure, equality is interpreted as meaning data flow. For example, the node $Input(B) = Output(A)$ describes the data flow into B from A. Also, the outcome of control flow splits is represented in the figure via the predicates $Succeeds(\cdot)$ and $Fails(\cdot)$.

The fringe of a dependency graph represents the supporting premises for the graph's root. These premises have been boxed in the figure, and are as follows: *Specification of A*, *Specification of B*, *Specification of Q*, which describe what *A*, *B* and *Q* compute; *Implementation of P*, which describes the control and data flow structure of *P*; *Specification of P*, which describes *P*'s specifications; and *Input(P)=0*, which is the initial test case for the program.

DEBUSSI forms an initial set of suspects from the premises that underlie the contradiction; in the figure, the suspects are enclosed in heavy boxes. By convention,

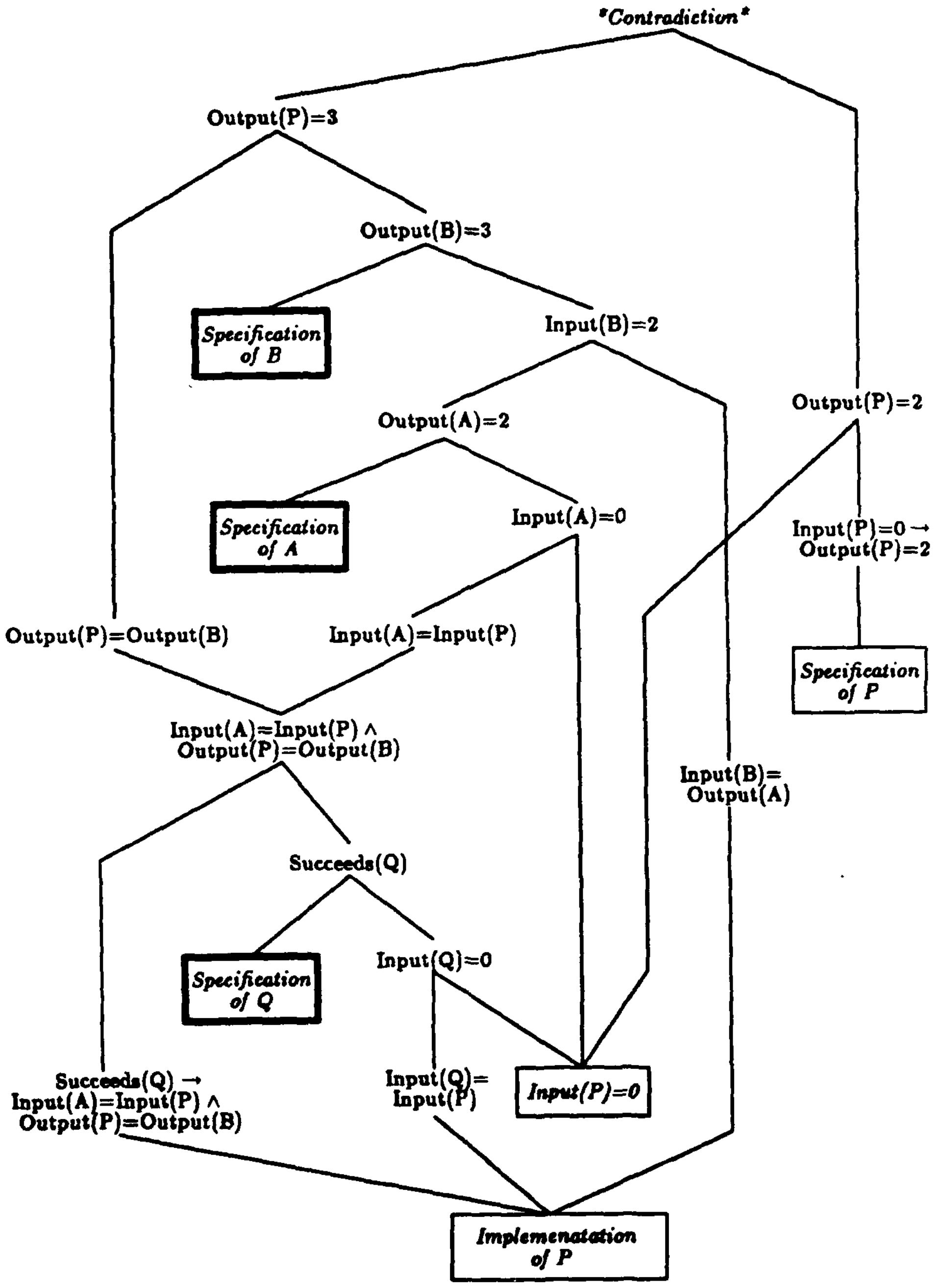


Figure 3.5: Dependency graph for a failed test case

some of the premises are disregarded as suspects; these are enclosed in light boxes in the figure. Some premises, such as the program's specification and test data, are disregarded because they are assumed to be "unbreakable." Other premises, such as the program's data flow, are disregarded as a way to simplify the localization task. (See Chapter 5 for more detail.) The initial suspects shown in Figure 3.5 are *Specification of A*, *Specification of B*, and *Specification of Q*.

Suppose that DEBUSSI queries the user about B, and the user replies B's argument is incorrect, i.e., $Input(B) \neq 2$. The result of this query is that DEBUSSI obtains new information about the *Implementation of P*, namely

$$Input(P) = 0 \rightarrow Input(B) \neq 2.$$

As shown in Figure 3.6, the information obtained via the query changes the bug manifestation. Prior to the query, the bug manifested itself at the output of P via the contradiction

$$Output(P) = 2 \text{ contradicts } Output(P) = 3.$$

As a result of the query, the bug manifests itself at the output of A via the contradiction

$$Output(A) = 2 \text{ contradicts } Output(A) \neq 2.$$

Because the bug manifestation has changed, the suspect set (premises underlying the manifestation) also changes. A comparison of Figures 3.5 and 3.6 illustrates how this change leads to the exoneration of B. The suspects which appear in Figure 3.6 are *Specification of A* and *Specification of Q*. Since *Specification of B* appears in Figure 3.5 but not in Figure 3.6, we have effectively exonerated B.

Now consider the effect of split analysis on control-flow splits in the program. Q is one such split, and the result of performing split analysis on it is shown in Figure 3.7. Split analysis results in a substantial change in the dependency graph, particularly in the absence of *Specification of Q* and the presence of *Specification of C*.

Earlier in this report, split analysis was presented as a process where DEBUSSI determines what would happen if a control flow split "went the other way." This is a view taken from the domain for programming. In terms of domain-independent debugging, split analysis is a technique whereby DEBUSSI coaxes the reasoning system to perform additional deduction and find alternate proofs.

In split analysis, DEBUSSI retracts the definition of a control flow split, and then asks the reasoning system to rederive the contradiction. (This rederivation is done via reasoning by cases on the outcome of the split.) If the contradiction can be demonstrated without the definition of a particular control flow split, then DEBUSSI can conclude that the split is not a cause for the bug.

Several nodes in the dependency graph reveal how DEBUSSI reasons about the possible outcomes of a split. For example,

$$Succeeds(Q) \oplus Fails(Q)$$

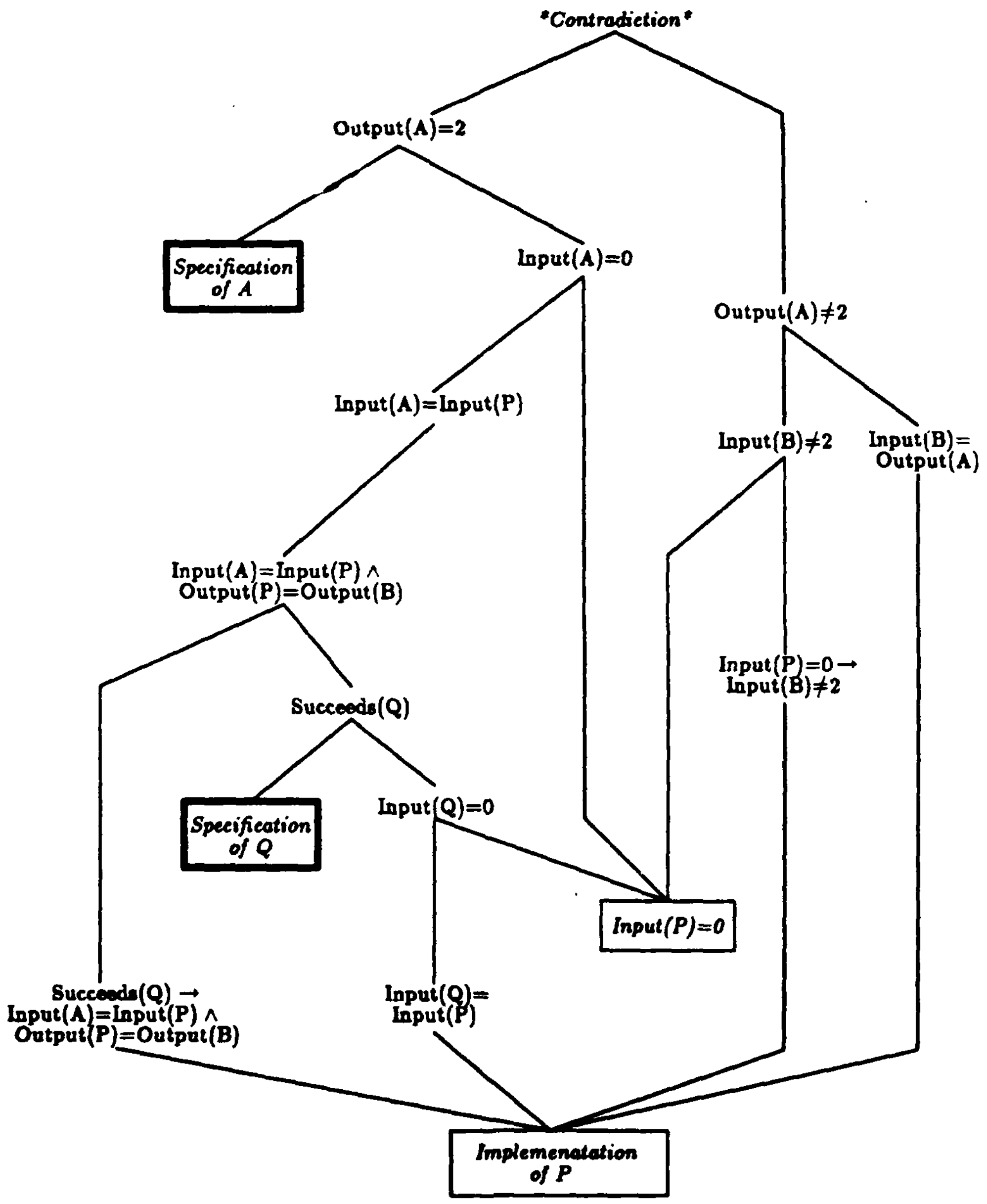


Figure 3.6: Dependency graph after query exonerates *B*

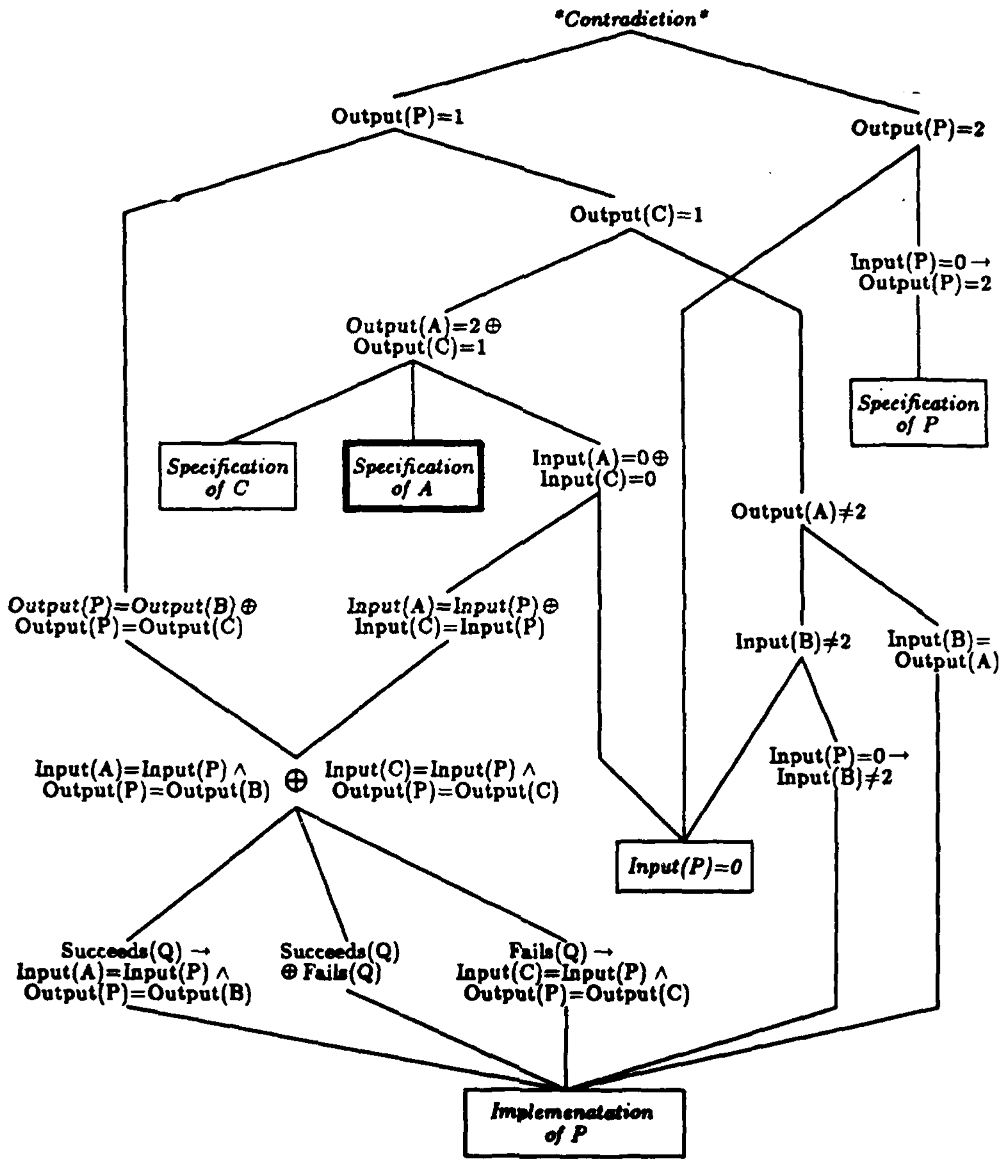

Contradiction
Output(P)=1
Output(P)=2
Output(C)=1
Input(P)=0 → Output(P)=2
Output(A)=2 ⊕ Output(C)=1
Specification of P
Specification of C
Specification of A
Input(A)=0 ⊕ Input(C)=0
Output(A)≠2
Output(P)=Output(B) ⊕ Output(P)=Output(C)
Input(A)=Input(P) ⊕ Input(C)=Input(P)
Input(B)≠2
Input(B)= Output(A)
Input(A)=Input(P) ∧ Output(P)=Output(B)
⊕
Input(C)=Input(P) ∧ Output(P)=Output(C)
Input(P)=0 → Input(B)≠2
Input(P)=0
Succeeds(Q) → Input(A)=Input(P) ∧ Output(P)=Output(B)
Succeeds(Q) ⊕ Fails(Q)
Fails(Q) → Input(C)=Input(P) ∧ Output(P)=Output(C)
Implemenatation of P

Figure 3.7: Dependency graph after split analysis exonerates Q

describes how the test Q can either succeed or fail, but not both. One consequence of this fact is

$$Output(P) = Output(B) \oplus Output(P) = Output(C),$$

which describes how the output of P is ultimately computed by B or by C.

As shown in Figure 3.7, the definition of Q is not required in order to bring about a contradiction at the output of P. Furthermore, the contradiction which arises is different from the initial contradiction shown in Figure 3.5.

Notice that C appears as a premise supporting the contradiction, yet is not added to the suspect set. This is because C's role is to support the notion of Q "going the other way." If Q had failed instead of succeeded, C would have been executed, thus C appears in the dependency graph.

Since a contradiction arises independently of the outcome of Q, Q can be exonerated, with the end result that DEBUSSI localizes the bug to A. The dependency graph in Figure 3.6 indicates that both $Input(A) = 0$ and $Output(A) \neq 2$ depend only on the specifications, implementation and test case for P. This means no further questions need to be asked about A, since there is already enough information available to prove that A was called correctly and returned the wrong answer. Thus DEBUSSI will conclude that the bug must be in the call to A.

3.4 Multiple Bugs

DEBUSSI adopts a multi-pass strategy for localizing multiple bugs. The goal of a single debugging session is to localize *one* bug. As discussed in Chapter 1, DEBUSSI repeatedly performs dependency analysis, split analysis and querying until the suspect set is either empty or a singleton. If a single suspect remains, and it corresponds to a call to a language primitive, then DEBUSSI concludes that that call is a bug. If a single suspect remains, and it corresponds to a call to a user-written function, then DEBUSSI expands the call and attempts to find a bug inside it.

If the suspect set is empty, then the program may contain multiple bugs (or it may contain buggy data flow or control flow). Multiple bugs can cause an an empty suspect set because of the way DEBUSSI performs split analysis. Programs that have more than one buggy split cause split analysis to fail by exonerating all the buggy splits.

Note that the process of considering multiple bugs has not been incorporated into the current implementation of DEBUSSI. When the suspect set is found to be empty, DEBUSSI will conclude that a bug is at some undetermined place within the enclosing function call, i.e., the function call that was last expanded in-line.

If DEBUSSI is to consider multiple bugs, any time the suspect set is found to be empty the localization process should begin anew with the goal of localizing two bugs. If the attempt to find a pair of bugs also yields an empty suspect set, then the process should repeat with the goal of localizing three bugs. Increasing numbers of bugs should be considered until the number of bugs equals the number of (possibly buggy) function calls.

One justification for DEBUSSI to avoid considering multiple bugs is the cost involved. For example, localizing a pair of bugs among n suspects requires testing all possible pairs that can be chosen out of those n, or

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = O(n^2)$$

combinations. Furthermore, the cost of split analysis grows exponentially in the number of bugs. Considering b bugs requires permuting the outcomes of b splits, and there are 2^b such permutations.

Another justification for focusing on one bug at a time is the observation that most multiple bug situations are "factorable" into several independent single bugs. In other words, we believe that programs where several bugs interact to hide each other's presence (such as Both-Odd) are relatively rare. Thus DEBUSSI attempts to localize a single "factor" of a multiple bug, knowing the tradeoff between the probability of encountering an unfactorable bug and the cost of having to localize it.

```
(Defun Both-Odd (X Y)
  (If (EvenP X)
      (If (EvenP Y)
          T
         Nil)
    Nil))
```

To illustrate how split analysis causes DEBUSSI to arrive at an empty suspect set, consider the buggy Both-Odd function shown above. Both-Odd is supposed to return T when both X and Y are odd, but instead, it returns T when both are even. Both-Odd has two bugs, namely that both calls to EvenP should be calls to OddP.

Recalling that DEBUSSI initially seeks to localize a *single* bug, consider what happens when we execute (Both-Odd 3 5), expecting T as an answer. Dependency analysis would first conclude that only suspect is the first call to EvenP. This returns Nil, causing the program to return Nil. During split analysis, the first call to EvenP would be made to return T, resulting in the execution of the second call to EvenP. This second call to EvenP would then return Nil, causing the incorrect value of Nil to be produced by the program. Thus split analysis on the first call to EvenP would cause it to be exonerated. Since this call to EvenP was the only suspect, the suspect set is now empty.

Another issue which influences the treatment of multiple software bugs is the difference between a function's definition and a function's usage. A function can be buggy because it is implemented incorrectly, or it can be buggy because it is the wrong function for the given task. Complications arise when a function having a single bug in its implementation is called several times by a program. The buggy function may produce a bug manifestation whenever it's called, thereby causing the program which uses it to appear to have several bugs.

Furthermore, given a set of bug manifestations, it is difficult to determine whether they are all attributable to a single implementation error or several usage errors. For example, suppose that three bug manifestations are found in a program, and they all occur at calls to `Func`. If `Func` is implemented incorrectly, then the program has exactly one bug. But if the first, second and third calls to `Func` should have really been calls to `Foo`, `Bar` and `Baz`, then the program has three bugs.

What saves DEBUSSI from these complications is that discerning between implementation and usage errors can be delayed until repair-time. For the purposes of bug localization, it is safe to assume that all bugs are in usage, i.e., all function calls are independent of each other. Once a particular call to function has been found to be buggy, the user can choose between repairing that particular call and redefining the function's definition.

3.5 Query Selection Heuristics

Queries are a way for DEBUSSI to obtain new specification information. Since the primary goal of DEBUSSI is to localize the bug as quickly as possible, the number of queries should be kept to a minimum. This need to minimize queries opposes the need to avoid overtaxing the user, because reducing the number of queries means increasing the amount of information that must be obtained per query. DEBUSSI employs a variety of heuristic methods in an attempt to find a middle-ground.

The remainder of this section describes the heuristics used by DEBUSSI for choosing queries. Heuristics are presented in order of their empirically determined usefulness. Note that these heuristics need not be used independently of one another. In fact, the current implementation of DEBUSSI uses a combination of them.

3.5.1 Binary Search

This heuristic implements the familiar "divide and conquer" concept. In binary search, the suspect chosen for the query one which lies near the "middle" of the partial order determined by the program's data flow. The logic behind this is that if the suspect is found to be called correctly, i.e., if its precondition is satisfied, then everything which precedes it by data flow can be exonerated. Thus a middle suspect has the potential to halve the number of suspects.

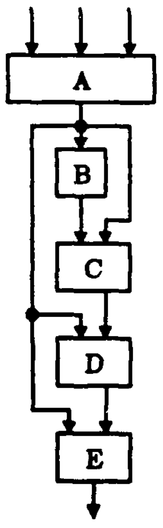
The strength of this heuristic is that it will localize the bug in time logarithmic in the number of executed program steps. The weakness of it is that it presents the user with scattershot questions that may cause confusion. For example, DEBUSSI may ask a question which, in the user's mind, very nearly localizes the bug. The next query may seem to break the chain of thought established by the first, leading the user to believe that DEBUSSI is confused. Or, even worse, the next query may involve a function call that is so deeply "buried" within the program that the user will be completely unable to answer the question.

3.5.2 Avoid Language Primitives

Programmers are usually protected from details of how language primitives are implemented. Thus when a call to a language primitive is found to be buggy, the only way to repair it is to replace it with a call to some other function. In other words, it is usually pointless to consider situations such as "Append has been implemented incorrectly."

The purpose of this heuristic is to bias DEBUSSI towards finding implementation errors in user-written code. Since language primitives are neither user-written nor subject to implementation errors, they should be avoided when choosing queries.

3.5.3 Maximal Data Flow Cover



This heuristic favors suspects that are most dependent on other suspects. For example, a suspect that accepts a large number of arguments and whose output fans out to many other suspects would be a good query candidate according to this heuristic. Each data flow arc introduces a dependency between the data's producer and consumer. Thus suspects with the most data flow throughput introduce the most new dependencies.

This approach is greedy, because its maximization strategy is based solely on local information about each suspect. It therefore has the potential to choose a query that appears good locally but is actually a poor choice globally. For example, in the plan shown to the left, the maximal data flow cover heuristic would favor A because its fanout of 4 represents the highest data flow throughput in the plan. But in fact A is not a good choice, because no suspect in the plan precedes it. In other words, A is a poor choice because it doesn't depend on any other suspects.

3.5.4 Sequential Stepping

This heuristic is familiar to any programmer who has used conventional debugging tools. Sequential stepping means examining suspects in the order in which they are executed. More generally, it means examining suspects in the order defined by their data flow structure. In DEBUSSI, this heuristic is used as a tie-breaker in sections of code where no suspect looks any more desirable than another.

The advantage of this approach is that it asks the user questions that follow an obvious (though naive) train of thought, thereby avoiding the scattershot questions which plague the binary search heuristic. The disadvantage of this approach is that it doesn't do anything that the user couldn't also do himself, e.g., with a conventional stepper.

Chapter 4

Related Work

4.1 Hardware Troubleshooting

This research was initially motivated by the apparent similarity between hardware troubleshooting and software bug localization. In hardware troubleshooting, the task is to find the failed component in a circuit that accounts for some observed misbehavior [4]. We believed that strategies that are useful for hardware troubleshooting could be applied to bug localization.

There are three important differences between circuits and programs. First, unlike circuits, programs rarely come with complete specifications of their behavior. Second, unlike circuits, the cost of probing internal values of programs is low [6]. As the following discussion will reveal, these two differences will balance each other.

A third difference between circuits and programs stems from the difference between a function *call* and a function *definition*. When a function call is found to be buggy, repairing it may either involve replacing it with a different function call, or redefining the function. In circuits, the only way to repair a buggy components is to replace it, i.e., one cannot "redefine" an AND gate.

In the following discussion, "device" will denote some component of a circuit or program. For example, an AND-gate is a device, as is a square-root subroutine. "Wiring" will denote the connection between two devices via data flow (in programs) or physical wires (in circuits). (Wiring is simply the kind of dependency that circuits and programs have in common). A "network" is one more devices that are wired together.

4.1.1 The Role of Specifications

A bug manifestation is a disagreement between specifications and observations. Typical hardware bug manifestations are *"Voltage at Y is not 5V"* or *"Resistor R7 is burning."* There is an obvious parallel between these and software bug manifestations like *"The value of Y is not 5.0"* or *"Program bombed in APPEND."*

Automated hardware troubleshooting techniques assume the presence of detailed specifications. Hardware specifications are typically given as simulation and inference rules. Simulation rules make forward deductions: they allow the outputs of a device to be determined from its inputs. Inference rules make backward deductions: they allow one or more inputs of a device to be determined from its outputs and other inputs.

Software debugging techniques cannot assume that a program is well specified. A program may be unfamiliar, providing no specifications other than its behavior when executed. Or a program may be only partially specified, describing behavior in a broad sense, but omitting details that are needed to find bugs.

Bugs are easier to detect in the presence of detailed specifications, because having more specifications means increasing the likelihood that some specification can be violated. This would seem to imply that it's easier to detect bugs in circuits, since they are usually more completely specified than programs. Unfortunately, in circuits, the leverage supplied by increased specification information is offset by the difficulty of making observations.

4.1.2 The Cost of Internal Probing

Values at the inputs and outputs of components of circuits or programs are determined by probing. In hardware, probing denotes the physical act of placing an instrument in contact with a wire in the circuit. In software, probing denotes the installation of instrumentation code to record values.

The amount of effort required to perform a probe differs dramatically between circuits and programs. Due to physical constraints, probing circuits is often very expensive, if not impossible. For example, several components of a circuit may lie within a single physical package, effectively shielded from observation. Also, a circuit board may be deeply buried within a chassis, preventing a probe from making physical contact.

The cost of probing software is more mental than physical, because unlike circuits, programs are relatively free of physical constraint. Physically, editing a low level subroutine is just as easy as editing a top level control loop. But mentally, editing low level routines requires the programmer to consider implementation details he would rather accept on faith.

4.2 Hardware Troubleshooting Approaches

This section discusses the hardware troubleshooting strategies that inspired the design of DEBUSSI. In the course of the discussion, differences between circuits and programs are shown to influence the applicability of these strategies to software bug localization.

Davis [4] proposes a methodology for diagnosing circuits based on reasoning about their structure and behavior. Davis' approach introduces the notion of dependency tracing as a way to determine suspects, and the technique of probing intermediate values as a way to collect new information. Also in this approach is *constraint suspension*, a technique whereby suspects are automatically exonerated.

Probes in Davis' system are analogous and complementary to queries in DEBUSSI. The difference in character between probes and queries is directly attributable to the availability of intermediate results in a program, and to the lack of specifications in a program. Davis' system probes intermediate circuit values to collect new *observations*, while DEBUSSI queries the user to collect new *specifications*. Davis' system compares its new observations to easily obtained specifications, while DEBUSSI compares its new specifications to easily obtained observations.

Davis' system has a technique for automatically exonerating suspects, called constraint suspension, where one explores the hypothesis that a circuit component is broken by retracting all the rules that govern its behavior. If the hardware diagnosis machinery is able to demonstrate a fault in the system without knowing how that particular component works, then the component can be exonerated.

Constraint suspension is incorporated in the process of split analysis, where DEBUSSI explores the hypothesis that a control flow split is buggy. The first step in split analysis is retracting the specification of the control flow split. Next, the reasoning system is asked to find an alternate derivation of the bug manifestation by "reasoning by cases" about the split's outcome. If an alternate derivation can be found, then the split can be exonerated.

The underlying goal of constraint suspension and split analysis is to coax an automated reasoning system to find an alternate, smaller support set for a bug manifestation. To find this new support, the reasoning system will often need detailed specifications. In hardware, these specifications abound, so constraint suspension is applicable to virtually any component. In software, specifications are limited, so constraint suspension can only be applied to functions that have very clear-cut characteristics. One such example is control flow splits, which all share the characteristic that they must either succeed or fail.

4.3 Overview of Software Debugging

Automatic program debugging has been an active area of research in Artificial Intelligence. The design of new debugging systems (this research included) is in part inspired by the successes and failures of old debugging systems.

Debugging systems differ in the way they represent programs. Some systems operate directly on the syntax of the programming language, and are thus designated to be language dependent. Other systems attempt to model programs in a language independent way, via some graphical representation or logical formalism.

Debuggers also differ in the way they reason about programs. Experience-based systems have libraries of heuristics that describe how to find common bugs. Other systems reason from first-principles, using only knowledge about the program to find bugs.

Debugging systems typically perform one or more of the following tasks: program recognition, bug detection, bug localization, bug explanation and bug correction. These tasks are usually done in the order mentioned. Systems that worry more about program testing tend to perform fewer of these tasks (i.e., only recognition and detection). Systems that tutor students must perform all of these tasks.

DEBUSSI can be described in terms of these three aspects. DEBUSSI is language independent, by virtue of the Plan Calculus [14] representation. It reasons about programs from first principles via its use of simple constraints. DEBUSSI localizes bugs and has some ability to explain bugs.

Several criteria are used to evaluate debugging systems. The first is *generality*. An ideal debugging system should be able to detect many types of bugs. It should be able to understand a variety of programs. Finally, it should be able to understand alternate implementations of the same algorithm.

Another criterion is *degree of automation*. The user of the debugging system should be required to do as little work as possible. If the user is an expert programmer, he should not have to answer questions about mundane details of his program. And if the user is a student, he should not have to interpret cryptic error messages.

Some debugging systems claim *cognitive plausibility*. The way a debugging system models programs should somehow parallel the programmer's own mental model. For example, a debugging system that will be used by experts shouldn't model a program at the syntactic level, because experts rarely make syntactic errors (i.e., an expert in Pascal will rarely omit a semicolon). But a debugging system that will be used by students must view a program at least partly syntactically, since that's the way students view programs.

DEBUSSI can be evaluated by these criteria. It is general, because any program or bug can be expressed in terms of first principles. It is highly automated, in that control flow analysis and constraint suspension require no user interaction. And it is cognitively plausible, because everyone must resort to first principles when they have no experience.

4.4 Tutoring Systems

One application of automatic program debugging is the tutoring of novice programmers. Tutoring systems are usually experience based, i.e., they maintain a library of algorithm descriptions that serve as templates for correct student programs. A tutoring system will compare a student's code to the appropriate algorithm description, transforming one or the other to account for minor implementation differences.

If a student's program cannot be matched to the algorithm description, experience-based bug detection is invoked. One by one, a collection of bug experts, each knowing the symptoms and cure for a specific bug, examines the code. When able, an expert modifies the buggy code to correct the bug and allow matching to continue.

It is important that a tutoring system provide good explanations. A good explanation describes the cause of a bug rather than its symptom. If a student understands how a bug arises, he or she can learn how to program defensively and prevent the bug from appearing again.

A tutoring system should also have cognitively plausible model because it's not only debugging programs, it's debugging the mind of the student. Any bug in a program can be traced to a specific misunderstanding in the mind of the student. A good debugging model makes this relationship explicit.

In Ruth's system [20], *Program Generation Models (*PGMs*)* describe algorithms by modeling the decisions made in writing a program. A PGM is like a context free grammar. Where context-free grammars derive valid strings in a language, PGMs derive valid implementations of an algorithm. A recursive descent parser called the *Action List Matcher* attempts to match a program to a PGM.

When the Action List Matcher is unable to parse a section of code, a bug has been found. Some bugs, such as loops that repeat the wrong number of times, require only minor changes in the source code to be corrected. These bugs are heuristically detected and corrected, thereby allowing the parse to continue. Other bugs, such as missing control structures, can only be corrected by major changes to the source code. These bugs indicate either that the wrong PGM is being matched with the program, or that the program is grossly incorrect.

PGMs are not guaranteed to represent all possible implementations of a given algorithm. If a student has a syntactically mangled implementation that happens to work, Ruth's system might consider the program incorrect. And if the student devises some clever new implementation of an algorithm, the PGM might not be able to derive it.

Adam and Laurent's **LAURA** [1] represents algorithms as *program models*. A program model is a supposedly correct implementation of an algorithm. Program models are written by the teacher in a traditional iterative language (FORTRAN).

LAURA converts the student's program and the program model into labeled control-flow graphs. During this process, a variety of transformations are systematically applied to canonicalize graph structure. LAURA then compares the two graphs, applying additional transformations in an attempt to make the graphs as similar as possible. Finally, any remaining differences are diagnosed from a set of known errors.

LAURA does not suggest corrections for errors, nor does it refer to syntactic elements of the program in its error messages. Instead, it presents the program model along with an annotated transformed version of the student's program. Examples of the annotations LAURA provides are "Line 15 in program 1 is unidentifiable" or "Different conditions on the arcs coming from lines 10 and 109."

The actual utility of presenting the student with annotated rewritten programs is questionable. An inexperienced student may not be able to understand why the rewritten version of his program is more correct than the original. And the vague annotations LAURA provides do not tell enough about what is actually wrong with the code. The student would learn more from a message like "Bad initialization of variable N in line 3" than he would from "Undefined instruction in line 3."

Program models in LAURA share the same shortcomings as PGM's in Ruth's system. An implementation of an algorithm may be correct even though its structure differs significantly from the program model.

Murray's **TALUS** [12] combines heuristic and formal methods. Heuristic methods are used to recognize algorithms, to guess at the possible locations of bugs and to suggest corrections for bugs. Formal methods are used to verify the equivalence of program fragments, to detect bugs and to prove or disprove heuristic conjectures.

TALUS views all programs, either student or teacher written, as collections of functions. Functions have abstract features such as recursion type and termination conditions. The measure of similarity between two functions is the number of abstract features they share. The measure of similarity between two programs is a weighted sum of the similarities of their component functions.

The first stage of debugging in TALUS is algorithm recognition. TALUS performs a best first search through all known algorithms to find the one algorithm that is most similar to the student's solution. Transformations are applied to facilitate matching with algorithms that have several functional decompositions.

The second stage of debugging is bug detection. In this stage, functions are represented as binary trees, with internal nodes representing conditional tests and leaf nodes representing function terminations or recursions. The set of conditions which must be true to reach a given leaf node defines a test case for that node. TALUS supplies each test case to both the student's solution and the matched algorithm. If the resulting returned values do not agree, a bug has been found.

The third and final stage of debugging in TALUS is bug correction. Top level expressions in the student's code fragment are replaced with their counterparts in the teacher's algorithm. When the two code fragments are found to be functionally equivalent, the bug has been completely corrected.

Representing an algorithm as a collection of abstract properties has several advantages. Algorithms are matched on the basis of abstract nonsyntactic features, so syntactically unconventional implementations will always be recognized. The properties which describe programs are language independent; with the appropriate parsers, algorithms and solutions can be written in any programming language. Bug descriptions drawn from abstract properties can replace a symptom with its cause (i.e., a message like "The loop variable X has been incorrectly initialized" describes a cause, whereas "The DO loop over variable X repeats 1 time too many" describes a symptom).

Johnson and Soloway's **PROUST** [8] debugs programs by reconstructing the goals of the student and identifying the elements of the program that were meant to realize the goals. This process is claimed to correspond to the actual thoughts of the student as he or she writes a program.

PROUST uses *programming plans* to represent common implementation fragments, both correct and buggy. For example, the "counter plan" describes the code where a variable is assigned an initial value and then incremented within the body of a loop. Programming plans are founded on the theory that expert programmers reason in terms of familiar algorithmic fragments, as opposed to primitive language constructs.

A programming task can be broken down into subtasks. A *goal decomposition* of a program describes the hierarchical structure of its subtasks, how its subtasks interact, and the mapping of subtask goals to the plans which implement them. PROUST relates programs to goals by matching plans from the goal decomposition *to the program's code.*

A problem description in PROUST can give rise to many correct and incorrect implementations. The initial description of the problem may have several goal decompositions, and each subtask in a goal decomposition may be implemented by several different plans.

PROUST avoids searching through all implementations of a task by using heuristics that describe which plans and goals will occur together. Thus goals are decomposed at the same time as plans are analyzed. As PROUST begins to understand a program, it establishes expectations to confirm its current line of reasoning. When an expectation fails, PROUST tries an alternate interpretation for the program.

PROUST is most useful as a tutoring tool. By attempting to capture the cognitive processes in program synthesis, it can assist a misguided student by appealing to his or her deeper understanding of program design. This is a feature missing in LAURA or TALUS, which simply present the bug in the code and suggest a repair.

4.5 Debugging Systems

Daniel Shapiro's **Sniffer** system [21] uses expert knowledge about programming to understand specific errors. Sniffer recognizes programs by identifying familiar algorithmic fragments, or *programming cliches* [14, 17, 24] in the code. Knowledge about bugs is encoded in bug experts, which generate detailed reports about errors.

A debugging session in Sniffer proceeds as follows. The user asks Sniffer to execute his program. As the program runs, Sniffer constructs an execution history containing information about when and where variables were modified, and what paths of control flow were taken.

The user interrupts execution at the first sign of trouble. He uses a *time rover* to search the program's execution history for bug symptoms and to localize the bug to a particular section of code. Once the location of the bug has been found, the programmer asks the *sniffer system* for a report.

The sniffer system performs two functions. First, it employs a *cliche finder* to recognize the familiar parts of the buggy code. Then bug experts are invoked to determine the exact nature of the bug. Bug experts use the time rover to verify symptoms for the bugs they specialize in.

Finally, Sniffer produces a detailed report about the bug. This report summarizes the error, analyzes the intended function of the code, and discusses how the bug manifested itself at runtime.

An advantage of Sniffer is its well defined modularity. In theory, one could easily augment the knowledge base of either the cliche finder or the sniffer system to suit any domain of possible bugs.

Because Sniffer is not given any specification information, it can neither detect nor localize bugs. This places unreasonable demands on the user, especially in large software systems. Bugs can manifest themselves in subtle ways in large systems, making their detection difficult. The number of components in a large system complicates the task of tracing a bug to its source.

Lukey's **PUDSY** [11] understands a program by building a description of the program. These descriptions can be compared to specifications to find bugs. Bugs occur where descriptions disagree with specifications.

Building a program description in PUDSY proceeds as follows. First, the program is grouped into *chunks*. A chunk is a schema for a common computation. A common type of chunk is a loop that finds an array's maximum element. PUDSY determines the data flow in and out of each chunk, and the data flow between chunks.

Next, PUDSY looks for *debugging clues* by using constraints on what "rational" programs look like. One such constraint is that a variable rarely appears in the left hand side of two consecutive assignment statements. Another constraint is that variable names are meaningful: a variable named **min** usually finds a minimum element. Violations of these constraints are usually noted for later use, but in some instances they can be used to immediately debug a section of code.

Describing a program in PUDSY is viewed as a stepwise process, where each step performs some transformation on the current description. An initial description of a chunk is made by trying to recognize it as an instance of a known schema. Every schema that can be recognized by PUDSY comes with a logical assertion that describes it. Assertions are combined by reasoning about program control and data flow. For example, a chunk that appears in the body of a loop can be quantified over the loop variable.

If the final program description does not agree with its specification, a bug has been found. PUDSY applies *backtracing* to determine the source of the bug. In backtracing, the inverse of each description-building transformation is applied to the program's specification. For example, if a transformation quantified an assertion in the description, backtracing would remove the equivalent quantification from the specification. In this way PUDSY can find the first point where descriptions and specifications disagree.

PUDSY's methodology for finding bugs is useful and reliable. Comparing complete specifications to descriptions will always find a bug if there is one. And looking for discourse clues in variable names is good way to detect low level differences between what the programmer meant to do and what he did by mistake.

Ehud Shapiro's system [22] debugs Prolog programs from first principles. Shapiro's system is very similar to this research in its use of first-principles reasoning. Three types of bugs are considered by his system: *termination with incorrect output*, when the output value of a deterministic procedure is incorrect; *finite failure*, when none of the outputs of a nondeterministic procedure are correct; and *nontermination*, when the program enters an infinite loop.

A debugging session in Shapiro's system consists of a question and answer session with the user. If some input causes a program to terminate with incorrect output, the system will selectively ask the user about the correctness of intermediate results of the computation. An approach termed *divide and query* performs a binary search on the steps of the computation to quickly focus in on the source of the bug.

Debugging a finite failure condition proceeds in a similar way. In this case, since the buggy procedure is nondeterministic, the debugger asks the user to supply *all* known solutions to intermediate results (making what is called an *existential query*).

Nontermination is debugged in several ways. First, the program can be run with bounds on space or time, on the assumption that exceeding these bounds implies that the program does not terminate. Also, *well founded orderings* can be defined on a procedure. An example of a well founded ordering is that consecutive calls to a divide and conquer procedure have decreasing parameter size.

DEBUSSI differs from Shapiro's system in several ways. First, DEBUSSI localizes bugs by reasoning about dependencies, while Shapiro's system localizes bugs by reasoning about execution traces. This point is illustrated by the following example.

```
(Defun Wasted-Effort (X Y)
  (Setq Y (F (G (H Y))))
  (P X))
```

If **Wasted-Effort** returned an incorrect value, Shapiro's system would suspect the calls to F, G, H and P because they all were executed. The only true suspect, however, is P, because it's the only function that the output depends on. So by employing dependency-directed reasoning, DEBUSSI is able to keep a tighter focus on the set of possible suspects.

Another important difference between DEBUSSI and Shapiro's system is DEBUSSI's use of automated reasoning. One could argue that Shapiro's system, by virtue of being built on top of Prolog, also uses automated reasoning. But Cake, the reasoning system used by DEBUSSI, is more general-purpose than Prolog. For example, Cake's truth maintenance machinery allows DEBUSSI to perform split analysis on conditional control flow. Also, Cake's type system allows DEBUSSI to detect abstract type conflicts in programs. Neither of these two reasoning techniques are available as part of the basic machinery of Prolog.

A final difference between the two systems is how they might scale to large programs. DEBUSSI is able to reason about programs hierarchically, treating complex functions as black boxes whose internal structure may more may not be important. By exploiting this hierarchy, DEBUSSI is able to focus its automated reasoning on one function at a time, and is therefore better suited for debugging large programs.

Korel's **PELAS** [9] is a system that is very similar to DEBUSSI. In PELAS, bugs are localized by a combination of interactive queries and a dependency-directed error locating mechanism. In fact, the only feature that differentiates DEBUSSI from PELAS is DEBUSSI's use of a general-purpose automated reasoning system.

Korel characterizes three kinds of influences (or dependencies) between a program's instructions. These are *data influence*, *control influence* and *potential influence*. Data and control influence are identical to the ideas of data and control flow dependency presented in Chapter 3 of this report. Potential influence represents the sections of code that are bypassed by the unexecuted half of a control flow split.

PELAS localizes bugs by backtracing through the dependency network established by these three influences, starting at the observed incorrect program output. Each backtracing step acts to localize the bug to a point further "upstream" in the dependency network. Backtracing iterates until a single faulty program instruction has been found.

Differences between PELAS and DEBUSSI stem from the comparative power of their respective reasoning engines. For example, PELAS' "error-locating module" limits the information it can obtain to knowledge of whether particular variables are correct or not. Thus queries in PELAS can refer only to particular values of variables.

Contrast this with Cake, which supports abstract reasoning about program structure and behavior, thereby allowing queries to refer to arbitrary properties of the buggy program.

Another difference between PELAS and DEBUSSI is DEBUSSI's use of split analysis. Because of split analysis, DEBUSSI is able to ask the user fewer questions than PELAS, thereby localizing bugs more quickly.

N. K. Gupta and Seviora's **Message Trace Analyzer** [5] uses an expert systems approach for debugging real time processes. Each process is modeled by a finite state machine that interacts with other processes by sending messages. The system constructs a structured model of interprocess communication called the *context tree*. The construction of the context tree is done through a multilevel subgoaling process. Components of the context tree are tested for failure by heuristic rules and state machine simulation.

The approach taken in the Message Trace Analyzer does not lend itself to the general debugging of traditional serial software systems. Debugging based solely on interprocess communication is akin to a pure I/O based debugging approach. In a complex software system, simple I/O discrepancies could have many equally valid explanations. One needs to understand the internal behavior of a program (or at least how it can be decomposed into simpler parts) in order to debug it.

One promising feature in the Message Trace Analyzer is its separation of general knowledge of real time systems from specific domain knowledge (the domain being telephone switching systems). This parallels the need for a software debugger to separate first principles programming knowledge from the knowledge of specific algorithms. A debugger that can maintain both types of knowledge and can intelligently decide to use one or the other would be quite useful indeed.

Harandi's **Knowledge Based Programming Assistant** [7] is another expert-systems approach. In this system, heuristic information is used to find many compile-time and run-time errors with well-defined symptoms. These heuristics are specified as situation/action pairs. The situation specifies bug symptoms and program information, and the action describes probable causes for the error and possible cures.

The apparent intent of [7] is to present a description of the knowledge base structure and inference system operation. Unfortunately, none of the actual rules for debugging are presented in the work.

4.6 Related Work in The Programmer's Apprentice

Waters [25] observes that two approaches have been traditionally used in the verification and debugging process, testing and inspection. Both approaches have problems when a large system must be dealt with. The utility of testing is limited by the imagination of the programmer who designs the tests. If the programmer cannot envision some unexpected error condition, he will not devise a test for it. The power of inspection is limited by the complexity of subprogram interactions in a large system. A programmer that inspects code to verify its correctness might not have the insight to consider the interaction of two seemingly unrelated subroutines.

Constraint modeling is a third way to verify and debug programs. The program is modeled as a network of constraints. The choice of what aspects of the program to model and what constraints to use is left up to the programmer. By performing constraint propagation on this network, bugs can be found that might not be found using testing or inspection. Waters concludes that the three approaches of testing, inspection and constraint modeling are mutually orthogonal, and are best used together in system verification.

Constraint modeling is the primary strategy used by DEBUSSI and by most hardware troubleshooters. Constraints are given by the specifications of the components of the program and their interconnections. Bugs are found by detecting contradictions (discrepancies) in this constraint network. Testing is used as a secondary strategy, as a method for determining an initial set of contradictions.

Chapman's **Program Testing Assistant** [3] helps programmers develop and maintain program test cases. The programmer tests functions in his program by specifying an expression to execute on some test data, along with correct results and success criteria. Each test case is associated with the set of functions it verifies through a set of abstract features. If any of those functions change, the test is re-run. If a success criterion is not met the programmer is warned of the error.

Wills' **Program Recognizer** [26] applies flow-graph parsing to the recognition of programs as plans in the plan calculus. A program is first transformed into a surface plan by control and data flow analysis. Surface plans represent programs in terms of functional boxes, data flow, control flow and constraints (DEBUSSI represents programs as surface plans). This surface plan is then translated into an extended flow graph (a type of labeled, acyclic, directed graph) to better facilitate subsequent matching. Flow graphs are parsed against a library of common structures to determine familiar program fragments. During the parse the original graph may be transformed to eliminate constraint violations.

Program recognition has always been considered an integral part of debugging. Wills' program recognizer factors this task out of the debugging process, allowing future research to concentrate more on bug localization and correction.

Levitin [10] explores the meaning and uses of errors in programming. A roughly day-long coding assignment in CLU (a strongly typed high-level language) was given to several volunteers. Versions of the program files were examined after the completion of the project to determine the quantity and nature of bugs encountered. Bugs were classified by such names as *missing guard*, *missing declaration*, and *malformed update*. Levitin concludes from this experiment that a general method for describing bugs is needed, one that works equally well for any programming task.

The method for describing bugs proposed by Levitin describes a bug as a vector in a space of categories. Each category has a metric associated with it that describes how the bug relates to that category. The categories chosen are *severity of error*, how serious the error is to the development process; *locus of error*, at what level of thought process the programmer erred; and *intent of error*, the realization of the programmer that a mistake was being made.

The metrics proposed vary depending on the category. Locus of error is measured across the spectrum from specification to implementation. Severity of error can be measured in amount of code changed, amount of time taken, or combinations of these and similar metrics. Intent of error is quantified based on the stage of the design process where the programmer decided to ignore some assumption about the program, and when the programmer realized the exact nature of the assumptions being violated.

Chapter 5

Limitations and Future Work

DEBUSSI has proven to be effective for localizing bugs in modestly sized programs. It utilizes several important techniques: reasoning about superficial properties of programs, manipulating dependencies in a truth maintenance system, and incremental acquisition of specification information.

DEBUSSI is not without its limitations. This chapter critically reexamines DEBUSSI, indicating where it's capabilities are limited. Then each limitation is addressed to illuminate how each one can be overcome. Finally, some extensions to DEBUSSI are discussed.

DEBUSSI is unable to localize errors in data flow. For example, suppose that a program calls Cons with its arguments reversed, and the buggy call is presented to the user as follows:

```
Function call: (Cons '(Apple Orange) 'Banana)
Returned (incorrect) value: ((Apple Orange) . Banana)

Was Cons called correctly? No
Enter violated conditions: Actual2=Arg1,
Enter violated conditions: Actual1=Arg2
                                              INTERACTION WINDOW
```

The user is only allowed to specify what values should have been supplied as arguments to Cons. As a result, DEBUSSI attempts to localize the bug to either the function that produced '(Apple Orange), or the function that produced 'Banana. But in fact neither of these functions are at fault, so DEBUSSI will only be able to localize the bug to the function call that encloses the call to Cons.

A first step towards allowing DEBUSSI to localize errors in data flow is to define some method for the user to discern between data *values* and data *flow*, and some logical predicates to describe common properties of data flow. With these extensions, the query presented above would appear as follows:

```
Function call: (Cons '(Apple Orange) 'Banana)
Returned (incorrect) value: ((Apple Orange) . Banana)

Was Cons called correctly? No
Enter violated conditions: DFlow(Source(Value2),Arg1),
Enter violated conditions: DFlow(Source(Value1),Arg2)
                                              INTERACTION WINDOW
```

In this hypothetical extension to DEBUSSI, the user utilizes the predicates DFlow and Source to specify, for example, that Arg1 should be accepting data flow from the same source as Value2.

DEBUSSI is limited by its query heuristics, which are too arbitrary and empirical. Some of the arbitrariness is revealed through the negative qualities of the various heuristics. For example, "binary search" has the tendency to ask scattershot questions, and the "avoid language primitives" tends to perform poorly in the case of simple "typographical error" bugs.

One possible way to improve DEBUSSI's query-finding ability is to give it more domain knowledge, such as a library of *programming clichés* [19]. A programming cliché is a familiar algorithmic fragment which can be compared to sections to the buggy program the detect a bug. (This is the debugging approach taken by Shapiro in the SNIFFER system [21].)

In choosing queries, it would also be useful to ask the user questions that are phrased at the proper level of abstraction and in the proper language for the domain. For example, suppose that while debugging a numerical program, DEBUSSI queries the user about some subprogram *Foo*, which exhibited the behavior *Foo(2)=3*. Rather than asking if 3 is the correct output for input 2,, DEBUSSI could ask about some abstract property that 3 satisfies. For example, DEBUSSI could ask if the output of *Foo* should be odd, or if the output should be prime, or if the output should be greater than the input. Finding appropriate properties can be done by inspecting conditional statements in the program, e.g., if the program uses EvenP, then DEBUSSI should ask if numerical values are even.

This ability would be desirable because it would improve the quality of specification information obtained. *Foo(2)=3* is indeed a partial specification for *Foo*, but it's only appropriate for the situations when *Foo's* input is 2. But *Odd(Output(Foo))* is appropriate for all inputs to *Foo*, and is thus a more desirable specification.

Although Lisp was chosen as the initial target language, DEBUSSI lacks an abstract model of Lisp semantics. The lack of this model is the reason why DEBUSSI requires programs to be side-effect free.

We believe that incorporating a more complete model of Lisp semantics will not change the nature of DEBUSSI's localization techniques. In fact, a more complete model of Lisp is certain to augment the kinds of dependencies that will underlie bugs, and will thereby increase the power of DEBUSSI's localization techniques. Such a model already exists, and it is presented in detail in [14]. Implementing this model is a current undertaking of the Programmer's Apprentice project.

DEBUSSI is limited by its single-fault assumption, and any further enhancement of DEBUSSI must involve expanding its repetoire to multiply-bugged programs. Several research directions need to be explored before this can happen. First, we need to better understand the difference between errors in a function's usage and errors in a function's definition. Second, we need to better understand the interactions between split analysis and other localization methods. Finally, we need to explore multiple bugs as they appear in a program's data flow and control flow.

A useful addition to DEBUSSI would be explanation capabilities. This means being able to justify a given query to the user during the debugging session, and providing a summary of the localization at the end of the session. Note that explanations can be read directly off the dependency structure for the bug manifestation. Formatting these explanations in a way that is meaningful to the user is nontrivial.

Perhaps the broadest long term goal is to integrate DEBUSSI into the Programmer's Apprentice. The user could edit programs using a knowledge-based editor [24], or could write source code by hand and then convert it to plans. Test cases, recalled from earlier stages of a program's design [3], would be used to elicit bug manifestations. After DEBUSSI localizes bugs, bug correction would be done with the assistance of a Design Apprentice [18].

The key issue in integrating a debugging tool such as DEBUSSI in the Programmer's Apprentice is determining the best use of program design knowledge. One possible use of design information has been alluded to already in the discussion of query selection. A programs's design will typically have information about its domain. This information can be used to influence the choice of queries, and to phrase queries in the language of the domain.

Bibliography

[1] Anne Adam and Jean-Pierre Laurent. LAURA, A System to Debug Student Programs. *Artificial Intelligence*, 15(1):75–122, November 1980.

[2] Daniel Brotsky and Charles Rich. Issues in the Design of Hybrid Knowledge Representation and Reasoning Systems. In *Proc. of the Workshop on Theoretical Issues in Natural Language Understanding*, Halifax, Nova Scotia, May 1985.

[3] David Chapman. A Program Testing Assistant. *Communications of the ACM*, 25(9), September 1982.

[4] Randall Davis. Diagnostic Reasoning Based on Structure and Behavior. AI Memo 739, MIT Artificial Intelligence Laboratory, June 1984.

[5] N. K. Gupta and R. E. Seviora. An Expert System Approach to Real Time System Debugging. In *Proceedings of the First Conference on Artificial Intelligence Applications*. IEEE Computer Society Press, December 1985.

[6] Walter Hamscher and Randall Davis. Issues in Model Based Troubleshooting. AI Memo 893, MIT Artificial Intelligence Laboratory, March 1987.

[7] Mehdi T. Harandi. Knowledge-Based Program Debugging: A Heuristic Model. In *Proceedings of SOFTFAIR*. IEEE Computer Society Press, July 1983.

[8] W. Lewis Johnson and Elliot Soloway. PROUST: Knowledge-Based Program Understanding. *IEEE Transactions on Software Engineering*, 11(3):267–275, March 1985. Reprinted in Rich, C. and Waters, R. C., editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufman, 1986.

[9] Bogdan Korel. PELAS - Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, 14(9):1253–1260, September 1988.

[10] Samuel M. Levitin. Toward a Richer Language for Describing Software Errors. MIT-AI Working Paper 270, May 1985.

[11] F. J. Lukey. Understanding and Debugging Programs. *International Journal on Man-Machine Studies*, 14:189–202, February 1980.

[12] William R. Murray. Heuristic and Formal Methods in Automatic Program Debugging. In *Proceedings of the IJCAI*, August 1985.

[13] Charles Rich. A Formal Representation for Plans in the Programmer's Apprentice. In *Proceedings of the IJCAI*, August 1981.

[14] Charles Rich. Inspection Methods in Programming. AI-TR 604, MIT Artificial Intelligence Laboratory, June 1981. PhD Thesis.

[15] Charles Rich. The Layered Architecture of a System for Reasoning about Programs. In *Proceedings of the IJCAI*, August 1985.

[16] Charles Rich. Inspection Methods in Programming: Cliches and Plans. AI-TR 1005, MIT Artificial Intelligence Laboratory, December 1987.

[17] Charles Rich and Richard C. Waters. Abstraction, Inspection and Debugging in Programming. AI Memo 634, MIT Artificial Intelligence Laboratory, June 1981.

[18] Charles Rich and Richard C. Waters. A Scenario Illustrating a Proposed Program Design Apprentice. AI Memo 933A, MIT Artificial Intelligence Laboratory, January 1987.

[19] Charles Rich and Richard C. Waters. The Programmer's Apprentice: A Research Overview. *IEEE Computer*, 21(11):10–25, November 1988. Reprinted in D. Partridge, editor, *Artificial Intelligence and Software Engineering*, Ablex, Norwood, NJ, 1989. Also published as MIT AI Memo 1004.

[20] Gregory R. Ruth. Intelligent Program Analysis. In *Readings in Artificial Intelligence and Software Engineering*, pages 431–441. Morgan Kaufmann, 1986.

[21] Daniel G. Shapiro. Sniffer: A System that Understands Bugs. AI Memo 638, MIT Artificial Intelligence Laboratory, June 1981. MS Thesis.

[22] Ehud Y. Shapiro. Algorithmic Program Debugging. Yale RR 237, Yale University, Department of Computer Science, May 1982. PhD Thesis.

[23] Jr. Steele, Guy Lewis. *Common LISP: The Language*. Digital Press, 1984.

[24] Richard C. Waters. KBEmacs: A Step Towards the Programmer's Apprentice. AI-TR 753, MIT Artificial Intelligence Laboratory, May 1985.

[25] Richard C. Waters. System Validation via Constraint Modeling. AI Memo 1020, MIT Artificial Intelligence Laboratory, February 1988.

[26] Linda M. Wills. Automated Program Recognition. AI-TR 904, MIT Artificial Intelligence Laboratory, February 1987. MS Thesis.

[27] Patrick H. Winston and Berthold K.P. Horn. *LISP*. Addison-Wesley, Reading, MA, third edition, 1984.